

Magnús M. Halldórsson (Ed.)

# Algorithm Theory – SWAT 2000

7th Scandinavian Workshop on Algorithm Theory  
Bergen, Norway, July 2000  
Proceedings



Springer



**Springer**

*Berlin*

*Heidelberg*

*New York*

*Barcelona*

*Hong Kong*

*London*

*Milan*

*Paris*

*Singapore*

*Tokyo*

Magnús M. Halldórsson (Ed.)

# Algorithm Theory – SWAT 2000

7th Scandinavian Workshop on Algorithm Theory  
Bergen, Norway, July 5-7, 2000  
Proceedings



Springer

**Series Editors**

Gerhard Goos, Karlsruhe University, Germany  
Juris Hartmanis, Cornell University, NY, USA  
Jan van Leeuwen, Utrecht University, The Netherlands

**Volume Editor**

Magnús M. Halldórsson  
University of Iceland and University of Bergen  
Taeknigardur, 107 Reykjavik, Iceland  
E-mail: mmh@hi.is

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Algorithm theory : proceedings / SWAT '2000, 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5 - 7, 2000, Magnús M. Halldórsson (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2000  
(Lecture notes in computer science ; Vol. 1851)  
ISBN 3-540-67690-2

**CR Subject Classification (1998): F.2, E.1, G.2, I.3.5, C.2**

**ISSN 0302-9743**

**ISBN 3-540-67690-2 Springer-Verlag Berlin Heidelberg New York**

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag is a company in the BertelsmannSpringer publishing group.  
© Springer-Verlag Berlin Heidelberg 2000  
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Steingräber Satztechnik GmbH, Heidelberg  
Printed on acid-free paper      SPIN: 10722125      06/3142      5 4 3 2 1 0

# Preface

The papers in this volume were presented at SWAT 2000, the Seventh Scandinavian Workshop on Algorithm Theory. The workshop, which is really a conference, has been held biennially since 1988, rotating between the five Nordic countries (Sweden, Norway, Finland, Denmark, and Iceland). It also has a loose association with the WADS (Workshop on Algorithms and Data Structures) conference that is held in odd numbered years. SWAT is intended as a forum for researchers in the area of design and analysis of algorithms. The SWAT conferences are coordinated by the SWAT steering committee, which consists of B. Aspvall (Bergen), S. Carlsson (Luleå), H. Hafsteinsson (U. Iceland), R. Karlsson (Lund), A. Lingas (Lund), E. Schmidt (Aarhus), and E. Ukkonen (Helsinki).

The call for papers sought contributions in all areas of algorithms and data structures, including computational geometry, parallel and distributed computing, graph theory, and computational biology. A total of 105 papers were submitted, out of which the program committee selected 43 for presentation. In addition, invited lectures were presented by Uriel Feige (Weizmann), Mikkel Thorup (AT&T Labs-Research), and Esko Ukkonen (Helsinki).

SWAT 2000 was held in Bergen, July 5–7, 2000, and was locally organized by a committee consisting of Pinar Heggernes, Petter Kristiansen, Fredrik Manne, and Jan Arne Telle (chair), all from the department of informatics, University of Bergen.

We wish to thank all the referees who aided in evaluating the papers. We also thank The Research Council of Norway (NFR) and the City of Bergen for financial support.

July 2000

Magnús M. Halldórsson

## Program Committee

Amotz Bar-Noy, Tel-Aviv Univ.  
 Luisa Gargano, Univ. of Salerno  
 Jens Gustedt, LORIA and INRIA Lorraine  
 Magnús M. Halldórsson, chair, U. Iceland and U. Bergen  
 Kazuo Iwama, Kyoto Univ.  
 Klaus Jansen, Univ. of Kiel  
 Jan Kratochvíl, Charles Univ.  
 Andrzej Lingas, Lund Univ.  
 Jaikumar Radhakrishnan, Tata Institute  
 R. Ravi, Carnegie Mellon Univ.  
 Jörg-Rudicher Sack, Carleton Univ.  
 Baruch Schieber, IBM Research  
 Sven Skjum, Univ. of Aarhus  
 Hisao Tamaki, Meiji Univ.  
 Jan Arne Telle, Univ. of Bergen  
 Esko Ukkonen, Univ. of Helsinki  
 Gerhard Woeginger, Technical Univ. Graz

## Referees

Pankaj Agarwal	Laura Grigori	Jiri Matoušek
Helmut Alt	Joachim Gudmundsson	Peter Bro Miltersen
Lars Arge	Bjarni V. Halldórsson	Gabriele Neyer
David Avis	Mikael Hammar	Anna Östlin
Luitpold Babel	Michael Houle	Rasmus Pagh
Bruno Beauquier	David Hutchinson	Jakob Pagter
Binay Bhattacharya	Rahul Jain	Christian N. S. Pedersen
Therese Biedl	Jesper Jansson	Marco Pellegrini
Andreas Björklund	Rolf Karlsson	Cecilia M. Procopiuc
Claudson Bornstein	T. Kavita	Wojtek Rytter
Gerth Stølting Brodal	Jyrki Kivinen	Fatma Sibel Salman
Eranda Cela	Rolf Klein	Sven Schuierer
Timothy M. Chan	Bettina Klinz	Eike Seidel
Joseph Cheriyan	Jochen Könemann	Pranab Sen
Johanne Cohen	Goran Konjevod	Jop Sibeyn
Artur Czumaj	S. Krishnan	Michiel Smid
Frank Dehne	Christos Levcopoulos	Edyta Szymanska
Ingvar Eidhammer	Moshe Lewenstein	Ariel Tamir
Aleksei V. Fishkin	Alex Lopez-Ortiz	Santosh Vempala
Éric Fleury	Ross McConnell	S. Venkatesh
Fedor Fomin	Ewa Malesinska	Alexander Wolff
Gudmund S. Frandsen	Monaldo Mastrolilli	Anders Yeo
Leszek Gąsieniec	Brian M. Mayoh	

# Table of Contents

## Invited Talks

- Dynamic Graph Algorithms with Applications ..... 1  
*Mikkel Thorup (AT&T Labs-Research) and David R. Karger (MIT)*

- Coping with the NP-Hardness of the Graph Bandwidth Problem ..... 10  
*Uriel Feige (Weizmann Institute)*

- Toward Complete Genome Data Mining in Computational Biology ..... 20  
*Esko Ukkonen (University of Helsinki)*

## Data Structures

- A New Trade-Off for Deterministic Dictionaries ..... 22  
*Rasmus Pagh (University of Aarhus)*

- Improved Upper Bounds for Pairing Heaps ..... 32  
*John Iacono (Rutgers University)*

- Maintaining Center and Median in Dynamic Trees ..... 46  
*Stephen Alstrup, Jacob Holm (IT University of Copenhagen),  
and Mikkel Thorup (AT&T Labs-Research)*

- Dynamic Planar Convex Hull with Optimal Query Time  
and  $O(\log n \cdot \log \log n)$  Update Time ..... 57  
*Gerth Stølting Brodal and Riko Jacob (University of Aarhus)*

## Dynamic Partitions

- A Dynamic Algorithm for Maintaining Graph Partitions ..... 71  
*Lyudmil G. Aleksandrov (Bulgarian Academy of Sciences)  
and Hristo N. Djidjev (University of Warwick)*

- Data Structures for Maintaining Set Partitions ..... 83  
*Michael A. Bender, Saurabh Sethia, and Steven Skiena  
(SUNY Stony Brook)*

## Graph Algorithms

- Fixed Parameter Algorithms for PLANAR DOMINATING SET  
and Related Problems ..... 97  
*Jochen Alber (Universität Tübingen), Hans L. Bodlaender (Utrecht  
University), Henning Fernau, and Rolf Niedermeier (Universität Tübingen)*

Embeddings of $k$ -Connected Graphs of Pathwidth $k$ . . . . .	111
<i>Arvind Gupta (Simon Fraser University), Naomi Nishimura (University of Waterloo), Andrzej Proskurowski (University of Oregon), and Prabhakar Ragde (University of Waterloo)</i>	

On Graph Powers for Leaf-Labeled Trees . . . . .	125
<i>Naomi Nishimura, Prabhakar Ragde (University of Waterloo), and Dimitrios M. Thilikos (Universitat Politècnica de Catalunya)</i>	

Recognizing Weakly Triangulated Graphs by Edge Separability . . . . .	139
<i>Anne Berry, Jean-Paul Bordat (LIRMM, Montpellier) and Pinar Heggernes (University of Bergen)</i>	

## Online Algorithms

Caching for Web Searching . . . . .	150
<i>Bala Kalyanasundaram (Georgetown University), John Noga (Technical University of Graz), Kirk Pruhs (University of Pittsburgh), and Gerhard Woeginger (Technical University of Graz)</i>	

On-Line Scheduling with Precedence Constraints . . . . .	164
<i>Yossi Azar and Leah Epstein (Tel-Aviv University)</i>	

Scheduling Jobs Before Shut-Down . . . . .	175
<i>Vincenzo Liberatore (UMIACS)</i>	

Resource Augmentation in Load Balancing . . . . .	189
<i>Yossi Azar, Leah Epstein (Tel-Aviv University), and Rob van Stee (Centre for Mathematics and Computer Science, CWI)</i>	

Fair versus Unrestricted Bin Packing . . . . .	200
<i>Yossi Azar (Tel-Aviv University), Joan Boyar, Lene M. Favrholdt, Kim S. Larsen, and Morten N. Nielsen (University of Southern Denmark)</i>	

## Approximation Algorithms

A $d/2$ Approximation for Maximum Weight Independent Set in $d$ -Claw Free Graphs . . . . .	214
<i>Piotr Berman (Pennsylvania State University)</i>	

Approximation Algorithms for the Label-Cover <sub>MAX</sub> and Red-Blue Set Cover Problems . . . . .	220
<i>David Peleg (Weizmann Institute)</i>	

Approximation Algorithms for Maximum Linear Arrangement . . . . .	231
<i>Refael Hassin and Shlomi Rubinstein (Tel-Aviv University)</i>	

Approximation Algorithms for Clustering to Minimize the Sum of Diameters .....	237
Srinivas R. Doddi, Madhav V. Marathe (Los Alamos National Laboratory), S. S. Ravi (SUNY Albany), David Scot Taylor (UCLA), and Peter Widmayer (ETH)	

## Matchings

Robust Matchings and Maximum Clustering .....	251
Refael Hassin and Shlomi Rubinstein (Tel-Aviv University)	

The Hospitals/Residents Problem with Ties .....	259
Robert W. Irving, David F. Manlove, and Sandy Scott (University of Glasgow)	

## Network Design

Incremental Maintenance of the 5-Edge-Connectivity Classes of a Graph ..	272
Yefim Dinitz (Ben-Gurion University) and Ronit Nossenson (Technion)	

On the Minimum Augmentation of an $\ell$ -Connected Graph to a $k$ -Connected Graph .....	286
Toshimasa Ishii and Hiroshi Nagamochi (Toyohashi University of Technology)	

Locating Sources to Meet Flow Demands in Undirected Networks .....	300
Kouji Arata (Osaka University), Satoru Iwata (University of Tokyo), Kazuhisa Makino, and Satoru Fujishige (Osaka University)	

Improved Greedy Algorithms for Constructing Sparse Geometric Spanners	314
Joachim Gudmundsson, Christos Levcopoulos (Lund University), and Giri Narasimhan (University of Memphis)	

## Computational Geometry

Computing the Penetration Depth of Two Convex Polytopes in 3D .....	328
Pankaj K. Agarwal (Duke University), Leonidas J. Guibas (Stanford University), Sariel Har-Peled (Duke University), Alexander Rabinovitch (Synopsys Inc.), and Micha Sharir (Tel Aviv University)	

Compact Voronoi Diagrams for Moving Convex Polygons .....	339
Leonidas J. Guibas (Stanford University), Jack Snoeyink (University of North Carolina), and Li Zhang (Stanford University)	

Efficient Expected-Case Algorithms for Planar Point Location .....	353
Sunil Arya, Siu-Wing Cheng (Hong Kong University of Science and Technology), David M. Mount (University of Maryland), and H. Ramesh (Indian Institute of Science)	

A New Competitive Strategy for Reaching the Kernel of an Unknown Polygon .....	367
<i>Leonidas Palios (University of Ioannina)</i>	

## Strings and Algorithm Engineering

The Enhanced Double Digest Problem for DNA Physical Mapping .....	383
<i>Ming-Yang Kao, Jared Samet (Yale University), and Wing-Kin Sung (University of Hong Kong)</i>	

Generalization of a Suffix Tree for RNA Structural Pattern Matching .....	393
<i>Tetsuo Shibuya (IBM Tokyo Research Laboratory)</i>	

Efficient Computation of All Longest Common Subsequences .....	407
<i>Claus Rick (Universität Bonn)</i>	

A Blocked All-Pairs Shortest-Paths Algorithm .....	419
<i>Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya (University of Florida)</i>	

## External Memory Algorithms

On External-Memory MST, SSSP, and Multi-way Planar Graph Separation .....	433
<i>Lars Arge (Duke University), Gerth Stølting Brodal (University of Aarhus) and Laura Toma (Duke University)</i>	

I/O-Space Trade-Offs .....	448
<i>Lars Arge (Duke University), and Jakob Pagter (University of Aarhus)</i>	

## Optimization

Optimal Flow Aggregation .....	462
<i>Subhash Suri, Tuomas Sandholm, and Priyank Ramesh Warkhede (Washington University)</i>	

On the Complexities of the Optimal Rounding Problems of Sequences and Matrices .....	476
<i>Tetsuo Asano (JAIST), Tomomi Matsui (University of Tokyo), and Takeshi Tokuyama (Tohoku University)</i>	

On the Complexity of the Sub-permutation Problem .....	490
<i>Shlomo Ahal (Ben-Gurion University) and Yuri Rabinovich (Haifa University)</i>	

Parallel Attribute-Efficient Learning of Monotone Boolean Functions .....	504
<i>Peter Damaschke (FernUniversität Hagen)</i>	

## Distributed Computing and Fault-Tolerance

Max- and Min-Neighborhood Monopolies .....	513
<i>Kazuhisa Makino (Osaka University), Masafumi Yamashita (Kyushu University), and Tiko Kameda (Simon Fraser University)</i>	
Optimal Adaptive Fault Diagnosis of Hypercubes .....	527
<i>Andreas Björklund (Lund University)</i>	
Fibonacci Correction Networks .....	535
<i>Grzegorz Stachowiak (University of Wrocław)</i>	
Least Adaptive Optimal Search with Unreliable Tests.....	549
<i>Ferdinando Cicalese, Ugo Vaccaro (Università di Salerno), and Daniele Mundici (Università di Milano)</i>	
<b>Author Index .....</b>	<b>563</b>

# Dynamic Graph Algorithms with Applications

Mikkel Thorup<sup>1</sup> and David R. Karger<sup>2</sup>

<sup>1</sup> AT&T Labs–Research, Shannon Laboratory, 180 Park Avenue, Florham Park, NJ 07932. [mthorup@research.att.com](mailto:mthorup@research.att.com).

<sup>2</sup> MIT Laboratory for Computer Science, Cambridge, MA 02138.  
[karger@theory.lcs.mit.edu](mailto:karger@theory.lcs.mit.edu)

**Abstract.** First we review amortized fully-dynamic polylogarithmic algorithms for connectivity, minimum spanning trees (MST), 2-edge- and biconnectivity. Second we discuss how they yield improved static algorithms: connectivity for constructing a tree from homeomorphic subtrees, 2-edge connectivity for finding unique matchings in graphs, and MST for packing spanning trees in graphs.

The application of MST for spanning tree packing is new and when boot-strapped, it yields a fully-dynamic polylogarithmic algorithm for approximating general edge connectivity within a factor  $\sqrt{2 + o(1)}$ .

Finally, on the more practical side, we will discuss how output sensitive algorithms for dynamic shortest paths have been applied successfully to speed up local search algorithms for improving routing on the internet, roughly doubling the capacity.

## 1 Dynamic Graph Algorithms

In this talk, we will discuss some simple dynamic graph algorithms and their applications within static graph problems. As a new result, we will derive a fully dynamic polylogarithmic algorithm approximating the edge connectivity  $\lambda$  within a factor  $\sqrt{2 + o(1)}$ , that is, the algorithm will output a value between  $\lambda/\sqrt{2 + o(1)}$  and  $\lambda \times \sqrt{2 + o(1)}$ .

The talk is not intended as a general survey of dynamic graph algorithms and their applications. Rather its goal is just to present a few nice illustrations of the potent relationship between dynamic graph algorithms and their applications in static graph problems, showing contexts in which dynamic graph algorithms play a role similar to that played by priority queues for greedy algorithms.

In a *fully dynamic graph problem*, we are considering a graph  $G$  over a fixed vertex set  $V$ ,  $|V| = n$ . The graph  $G$  may be *updated* by insertions and deletions of edges. Unless otherwise stated, we assume that we start with an empty edge set. We will review the fully dynamic graph algorithms of Holm et al. [11] for connectivity, minimum spanning trees (MST), 2-edge, and biconnectivity in undirected graphs. For the connectivity type problems, the updates may be interspersed by queries on (2-edge-/bi-) connectivity of the graph or between specified vertices. For MST, the fully dynamic algorithm should update the MST in connection with each update to the graph: an inserted edge might have to go into the MST, and if an MST edge is deleted, we should replace with the lightest edge possible.

Both updates and queries are presented *on-line*, meaning that we have to respond to an update or query without knowing anything about the future.

The time bounds for these operations are polylogarithmic but *amortized* meaning that we only bound the average operation time over any sequence of operations, starting with no edges. In our later applications for static graph problems, we only care about the total amount of time spent over all dynamic graph operations, and hence the amortized time bounds suffice.

The above mentioned results are all for undirected graphs. For directed graphs there are very few results. In a recent break-through, King [16] showed how to maintain the full transitive closure of a graph in  $\tilde{O}(n^2)$  amortized time per update. Further, she showed how to maintain all pairs shortest paths in  $O(n^{2.5}\sqrt{\log C})$  time per update if  $C$  is the maximum weight in the graph. However, if one is just interested in maintaining whether  $t$  can be reached from  $s$  for two fixed vertices  $s$  and  $t$ , nobody knows how to do this in  $o(m)$  time.

On the more practical side, Ramalingan and Reps [24] have suggested a lazy implementation of Dijkstra's [4] single source shortest paths algorithm for a dynamic directed graph. If  $X$  is the number of vertices that change distance from the source  $s$  in connection with an arc insertion or deletion, they can update a shortest path tree from  $s$  in  $\tilde{O}(\sum_{v \in X} \text{degree}(v))$  time. Although this does not in general improve over the  $\tilde{O}(m)$  time it takes to compute a single source shortest path tree from scratch, there has been experimental evidence suggesting that this kind of laziness is worthwhile in connection with internet like topologies [7].

## 2 Applications

We are now going to review some simple applications of dynamic graph algorithms for solving problems on static graphs.

### 2.1 Dynamic Connectivity and the Construction of Trees from Homeomorphic Subtrees

Our first application is of Henzinger, King, and Warnow [10]. We are given a set  $A$  of leaves and a set  $X$  of triples  $((a, b), c) \in A^3$ . The problem is then, if possible, to find a so-called *consensus tree*  $T$  with leaf set  $A$  such that for every  $((a, b), c) \in X$ , the least common ancestor of  $a$  and  $b$  is a strict descendant of the least common ancestor of  $b$  and  $c$ . The consensus tree problem was raised in 1981 by Aho, Sagiv, Szymanski, and Ullman in the context of optimizing relational expressions [2], and they presented an  $O(|A||X|)$  time solution. Since then the consensus problem has found applications in computational biology [10, 12]. Henzinger, King, and Warnow [10] have reduced the consensus tree problem to decremental connectivity, thereby getting an  $\tilde{O}(|X|)$  bound.

An example of an application of fully dynamic connectivity, due to Karger [4], is for identifying highly connected subgraphs in a randomized max-flow algorithm. This application is, however, too complicated for the current talk.

## 2.2 Dynamic 2-Edge Connectivity and Matchings

The dynamic algorithms for 2-edge connectivity have proved useful in making efficient constructions in relation to some classical theorems in matching theory.

A theorem of Petersen from 1891 [22] states that every bridgeless 3-regular graph has a perfect matching. Biedl, Bose, Demaine, and Lubiw [3] have used dynamic 2-edge connectivity to construct such a perfect matching in  $\tilde{O}(n)$  time, improving over the bound the  $\tilde{O}(n^{3/2})$  obtained using the general time bound for matching when  $m = O(n)$ .

A theorem of Kotzig from 1959 [17] states that any unique perfect matching contains a bridge. Using dynamic 2-edge connectivity for maintaining bridges, Gabow, Kaplan, and Tarjan [9] used Kotzig's theorem to check if a graph has a unique matching, improving the running time from  $\tilde{O}(mn)$  to  $\tilde{O}(m)$ .

In this talk we will only review the simple and elegant construction of Gabow, Kaplan, and Tarjan.

## 2.3 Dynamic MST, Tree Packing, and Edge Connectivity

The dynamic MST algorithm can be used directly to speed up  $(1-\varepsilon)$ -approximate tree packing based on the Lagrangian relaxation techniques suggested by Plotkin, Smoys, and Tardos [23], with the refinements of Young [25]. Using a theorem of Nash-Williams [21] this leads to a  $\sqrt{2 + \varepsilon}$ -approximation of the edge connectivity of a graph. What makes all this really interesting is that the construction can be made fully dynamic, implying that we can maintain the edge connectivity of a graph within a factor  $\sqrt{2 + o(1)}$  in polylogarithmic amortized time per operation.

The above construction is new and the details are presented in Section 3.

## 2.4 Dynamic Shortest Paths and Local Search for Routing on the Internet

Our last application is of practical nature, illustrating how dynamic graph algorithms can be used to speed up local search [1]. The general strategy in local search is to iteratively improve some feasible solution by considering small changes to it. In the context of graph, this small change may be the insertion or deletion of an edge, or just a weight change. We can then speed up the local search if we can find an efficient solution to the fully-dynamic graph problem of maintaining the objective function under edge updates and weight changes.

This general strategy was recently used in a local search for improving the capacity of the internet by Fortz et al. [6]. Currently, Open Shortest Path First (OSPF) [19] is the most widely used intra-domain internet routing protocol. Packets are routed along shortest paths to their destination. The weights of the links, and thereby the shortest path routes, can be changed by the network operator. The weights could be set proportional to their physical distances, but often the main goal is to avoid congestion, i.e. overloading of links, and the

standard heuristic recommended by Cisco is to make the weight of a link inversely proportional to its capacity.

A natural question is if one can improve the weight setting given some estimate of the demands. In [6] this was tested both for a proposed AT&T WorldNet backbone, and for various synthetic networks, and weight setting were found allowing for a 50–110% increase in the demands over what is achieved with standard weight setting heuristics.

The approach in [6] is a local search algorithm where one repeatedly tries to make one or a few weight changes, and see if this improves the routing. In order to simulate and evaluate the routing relative to these changes, we need to recompute all pairs shortest paths. The networks considered were all sparse ( $m \leq 4n$ ) so the  $\tilde{O}(n^{2.5}\sqrt{\log C})$  time algorithm of King [16] would be worse than recomputing from scratch in  $\tilde{O}(n^2)$  time. Instead we used the lazy approach of Ramalingam and Reps [24]. Even though the networks considered were comparatively small ( $n \leq 100$ ), this lead to a speed up from around 20 hours and down to about 1.5 hours, thus making the programs much more attractive to the business units at WorldNet.

### 3 Tree Packings and Edge Connectivity

We will now present the new results on tree packing and fully dynamic edge connectivity. The two concepts are strongly tied, as detailed below.

The *edge connectivity* of a graph is the minimal number of edges whose removal disconnects graph. We will denote this number by  $\lambda(G)$ , or just  $\lambda$  when  $G$  is understood. Note that  $\lambda n \leq 2m$  since the minimal degree is an upper bound on  $\lambda$ .

A *tree packing* in  $G$  is an assignments of weights to spanning trees of  $G$  so that each edge  $e$  has gets *load*

$$\ell(e) = \sum_{T:e \in T} w(T) \leq 1.$$

The value of the tree packing is  $\sum_T w(T)$ . We let  $\tau$  denote the value of the maximal tree packing of  $G$ .

**Theorem 1 (Nash-Williams [21]).**  $\lambda/2 \leq \tau \leq \lambda$ .

Above,  $\tau \leq \lambda$  follows directly from the fact that any cut is crossed by all spanning trees.

We are going to use fully dynamic MST to speed up a  $(1 - \varepsilon)$ -approximate tree packing based on Lagrangian relaxation [23][25], improving the running time from  $\tilde{O}(\lambda m)$  to  $\tilde{O}(m)$ . For comparison, the best exact tree packing of Gabow [8] takes  $\tilde{O}((\lambda n)^2)$  time.

Next, we will argue that this  $(1 - \varepsilon)$ -approximate tree packing can be maintained dynamically, using polylogarithmic amortized time per operation if  $1/\varepsilon$  and the edge connectivity is polylogarithmic. Rounding and multiplying our estimated packing value by  $\sqrt{2}$ , we will approximate the edge connectivity within a factor  $\sqrt{2}$ .

Applying the above construction to a logarithmic number of random subgraphs  $G(p)$ ,  $p = 1, 1/2, \dots$ , with high probability, we end up maintaining edge connectivity within a factor  $\sqrt{2 + o(1)}$  in polylogarithmic amortized time per operation.

For fully-dynamic connectivity, the best exact algorithm spends  $\tilde{O}(\lambda n)$  time per update, combining the  $\tilde{O}(m)$  randomized static edge connectivity algorithm of Karger [15] with the sparse certificates of Nagamochi and Ibaraki [20], and the dynamic sparsification technique of Eppstein et al. [5].

A previous dynamic randomized fully dynamic edge connectivity algorithm has been suggested by Karger [13]. For any  $\alpha > 0$ , it produces a  $\sqrt{1 + 2/\alpha}$ -approximation with  $\tilde{O}(n^\alpha)$  amortized update time if combined with the fully dynamic polylogarithmic MST technique from [11]. For example, with  $\alpha = 1/2$ , it uses  $\tilde{O}(\sqrt{n})$  amortized update time to get an approximation factor of  $\sqrt{5}$ , as opposed to our factor  $\sqrt{2 + o(1)}$  with polylogarithmic amortized update time.

### 3.1 Lagrangian Tree Packing

Young has verified [personal communication, 1999] that his variant [25] of the Lagrangian packing technique of Shmoys, Plotkin, and Tardos [23] implies the following result when specialized to tree packing:

**Theorem 2** (Shmoys et al. [23], Young [25]). *The following algorithm produces a tree packing of value  $W$  where  $(1 - \varepsilon)\tau \leq W \leq \tau$ .*

1. Initially no spanning tree has any weight.
2. Set  $W := 0$ .
3. While no edge has load 1
  - (a) Pick a load-minimal spanning tree  $T$ .
  - (b)  $w(T) := w(T) + \varepsilon^2 / (3 \ln m)$ .
  - (c)  $W := W + \varepsilon^2 / (3 \ln m)$ .
4. Return  $W$ .

Since each iteration increases the value of the tree packing by  $\varepsilon^2 / (3 \ln m)$ , the above algorithm must terminate in  $\tau 3 \ln m / \varepsilon^2 \leq \lambda 3 \ln m / \varepsilon^2$  iterations. Using a standard static MST algorithm, this takes  $\tilde{O}(\lambda m \ln m / \varepsilon^2)$  time.

Using the dynamic MST algorithm from [11], we get

**Theorem 3.** *A  $(1 - \varepsilon)$ -approximate tree packing can be constructed in  $\tilde{O}(m / \varepsilon^2)$  time.*

*Proof.* We use the dynamic MST algorithm from [11] to maintain the load minimal spanning tree in the algorithm from Theorem 2. In each iteration, we first make a copy  $T$  of the current spanning tree, and then, for each edge  $e \in T$ , we increase the load by  $\varepsilon^2 / 3 \ln m$ . Since no load gets beyond 1, the total number of load increases is  $O(m \log m / \varepsilon^2)$ , and each load increase is supported in  $\log^{O(1)} n$  amortized time.

By Theorem 11, if we multiply the found tree packing value by  $\sqrt{2}$ , we immediately get a deterministic  $\tilde{O}(m)$  time  $\sqrt{2 + o(1)}$ -approximation of edge connectivity, matching a result of Matula [18].

### 3.2 Fully-Dynamic Packing with Small Cuts

We are now going to present a fully dynamic version of the algorithm from the previous section, assuming that we are only interested in edge connectivity  $\leq \lambda_{\max}$ . Also, assume, for the moment, that the graph remains connected.

We will pack  $q = 3\lambda_{\max} \ln m / \varepsilon^2$  spanning trees  $T_1, \dots, T_q$ , each with weight 1. We will not have any limits on how much load can be put on the edges. Using the MST data structure from [11], each spanning tree  $T_i$  is a load minimal spanning tree where the load of edge  $e$  is

$$\ell_{i-1}(e) = |\{T_j : j < i, e \in T_j\}|$$

For technical reasons, we assign unique priorities to the edges so that edges of higher priority are preferred. Then the MST is unique. The priority of an edge is the same for all the different values of  $i$ .

The maximal load in our packing is

$$L = \max_{e \in E} |\{T_i : e \in T_i\}|$$

Scaling down the weights by  $L$  to give maximal load 1, we get a tree packing of value  $q/L$ .

If  $q/L > \lambda_{\max}$ , we conclude that the graph has edge connectivity  $\geq \lambda_{\max}$ . Otherwise, by Theorem 2,  $(1 - \varepsilon')\tau / \geq q/L \leq \tau$ , where  $\varepsilon'^2/(3 \ln m) = 1/L \leq \lambda_{\max}/q = \varepsilon^2/(3 \ln m)$ , so  $\varepsilon' \leq \varepsilon$ .

For the analysis of the update time of the above algorithms, we need the following lemma:

**Lemma 1.** *Each insertion or deletion of an edge changes at most  $i$  edge loads  $\ell_i$  for each  $i$ .*

*Proof.* By symmetry, it suffices to consider the deletion of an edge  $e$ .

Since we assumed the graph remains connected, the total load remains  $i(n - 1)$ . When  $e$  is deleted, its load drops to 0, so the load increase over all edges  $f \neq e$  is  $\ell_i^{old}(e) \leq i$ . Consequently, it suffices to prove that loads on edges  $f \neq e$  do not decrease. By induction over  $i$ , we may assume  $\ell_{i-1}^{new}(f) \geq \ell_{i-1}^{old}(f)$ .

Now, if  $f \notin T_i^{old} \setminus T_i^{new}$ ,  $\ell_i^{new}(f) - \ell_i^{old}(f) \geq \ell_i^{old}(f) - \ell_{i-1}^{old}(f)$ , so  $\ell_i^{new}(f) \geq \ell_i^{old}(f)$ , as desired. Suppose instead that  $f \in T_i^{old} \setminus T_i^{new}$ . Since the edges have unique priorities and no  $\ell_{i-1}$  load has decreased,  $f \in T_i^{old} \setminus T_i^{new}$  implies  $\ell_{i-1}^{new}(f) > \ell_{i-1}^{old}(f)$ . Hence  $\ell_i^{new}(f) \geq \ell_{i-1}^{new}(f) \geq \ell_{i-1}^{old}(f) + 1 \geq \ell_i^{old}(f)$ . This completes the proof of the lemma.

**Theorem 4.** *For graphs with edge connectivity  $\leq \lambda_{\max}$ , we can maintain a  $(1 - \varepsilon)$ -approximate tree packing in  $\tilde{O}(\lambda_{\max}^2 \text{polylog } n / \varepsilon^4)$  amortized time per update. The algorithm announces that the edge connectivity is  $> \lambda_{\max}$  before it gives a worse approximation.*

*Proof.* The time spent on maintaining the MST data structure from [11] for each  $T_i$  is amortized over load changes. Each load change takes polylogarithmic amortized time. Hence the result follows from Lemma 10 stating that the total number of load changes per edge update is bounded by  $\sum_{i \leq q} i < q^2$ .

If we remove the condition that the graph remains connected, each  $T_i$  should instead be a load minimal spanning forest, that is, a load minimal forest with a tree spanning each connected component. The algorithm from [11] maintains such minimal spanning forests. If the graph is disconnected, instead of returning  $q/L$ , we just return 0. Concerning the analysis, if a bridge  $e$  from some component is deleted/inserted, this simply has the effect of deleting/inserting  $e$  in each  $T_i$ , thus giving  $q$  dynamic MST operations. To analyze the deletion or insertion of a non-bridge, we simply apply the analysis from Lemma 10 to the affected component, thus getting the same  $\sum_{i \leq q} i < q^2$  bound as above.

**Corollary 1.** *For graphs with polylogarithmic edge connectivity, we can maintain the edge connectivity within a factor  $\sqrt{2}$  in polylogarithmic time per update, and further, announce if the graph is too connected for the approximation guarantee.*

*Proof.* Since the edge connectivity is an integer, we just set  $\varepsilon = 1/(6\lambda_{\max})$ , where  $\lambda_{\max}$  is polylogarithmic, and round the value of the tree packing to the nearest multiple of 1/2 to get a value  $W$  in  $[\lambda/2, \lambda]$ . Afterwards we return  $\sqrt{2}W$ .

### 3.3 Larger Edge Connectivity

The approach from the previous section gives polylogarithmic time bounds for polylogarithmic edge connectivity. In order to approximate larger edge connectivity, we consider random subgraphs of the graph in question. Let  $G(p)$  denote the random subgraph of  $G$  including each edge of  $G$  independently with probability  $p$ .

**Lemma 2 (Karger [13]).** *Let  $G$  be a graph with edge connectivity  $\lambda$  and let  $p\lambda \geq 6 \ln n / \varepsilon^2$ . Then the probability that the value of any cut in  $G(p)$  differs by a factor  $(1 + \varepsilon)$  from its expected value is  $O(1/n)$ .*

It is now easy to provide a  $\sqrt{2 + o(1)}$  approximation dynamic algorithm for edge connectivity:

- For  $p = 1, 1/2, 1/4, 1/8, \dots$ , let  $H_p = G(p)$ . That is, whenever an edge is inserted in  $G$ , it is inserted in  $H_p$  with probability  $p$ .
- For each  $H_p$ , maintain edge connectivity  $\leq \log^2 n$  as described in Corollary 10.
- After each edge update, let  $p_{\max}$  be the largest value of  $p$  for which the algorithm from Corollary 10 does not report that the edge connectivity is too large.
- Our approximate edge connectivity is the one approximated for  $H_{p_{\max}}$  divided by  $p_{\max}$ .

**Theorem 5.** *There is a randomized fully dynamic algorithm supporting updates in amortized polylogarithmic time that with high probability approximates edge connectivity within a factor  $\sqrt{2} + o(1)$ .*

*Proof.* Since we only maintain a logarithmic number of graphs  $H_p$ , the time bound is immediate from Corollary 1.

If  $\lambda = O(\log n)$ , we will have  $p_{\max} = 1$ , and hence we get a factor  $\sqrt{2}$  directly from Corollary 1. However, for  $\lambda = \omega(\log n)$ , the result is immediate from Lemma 2.

## References

1. E. H. L. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization*. Discrete Mathematics and Optimization. Wiley-Interscience, Chichester, England, June 1997.
2. A.V. Aho, Y. Sagiv, T.G. Szymanski, and J.D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM J. Computing*, 10(3):405–421, 1981.
3. T.C. Biedl, P. Bose, E.D. Demaine, and A. Lubiw. Efficient algorithms for Petersen’s matching theorem. In *Proc. 10th ACM-SIAM Symp. on Discrete Algorithms*, pages 130–139, 1999.
4. E. W. Dijkstra. A note on two problems in connection with graphs. *Numer. Math.*, 1:269–271, 1959.
5. D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification — a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997. See also FOCS’92.
6. B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *Proc. 19th IEEE INFOCOM - Conf. Computer Communications*, pages 519–528, 2000.
7. D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone. Experimental analysis of dynamic algorithms for the single-source shortest path problem. *ACM J. Experimental Algorithms*, 3, article 5, 1998.
8. H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *J. Comp. Syst. Sc.*, 50:259–273, 1995.
9. H.N. Gabow, H. Kaplan, and R.E. Tarjan. Unique maximum matching algorithms. In *Proc. 31st ACM Symp. on Theory of Computing*, pages 70–78, 1999.
10. M.R. Henzinger, V. King, and T. Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24(1):1–13, 1999.
11. J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proc. 30th ACM Symp. on Theory of Computing*, pages 79–89, 1998.
12. S. Kannan, T. Warnow, and S. Yooseph. Computing the local consensus of trees. *SIAM J. Computing*, 27(6):1695–1724, 1998.
13. D. R. Karger. Using randomized sparsification to approximate minimum cuts. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 424–432, 1994.
14. D. R. Karger. Better random sampling algorithms for flows in undirected graphs. In *Proc. 9th ACM-SIAM Symp. on Discrete Algorithms*, pages 490–499, 1998.
15. D. R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1), 2000.

16. V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40th IEEE Symp. on Foundations of Computer Science*, pages 81–89, 1999.
17. A. Kotzig. On the theory of finite graphs with a linear factor I. *Mat.-Fyz. Časopis Slovensk. Akad. Vied*, 9:73–91, 1959.
18. D. W. Matula. A linear time  $2 + \epsilon$  approximation algorithm for edge connectivity. In *Proc. 4th ACM-SIAM Symp. on Discrete Algorithms*, pages 500–504, 1993.
19. J. T. Moy. *OSPF: Anatomy of an Internet Routing Protocol*. Addison-Wesley, 1999.
20. H. Nagamochi and T. Ibaraki. Linear time algorithms for finding a sparse  $k$ -connected spanning subgraph of a  $k$ -connected graph. *Algorithmica*, 7:583–596, 1992.
21. C. St. J. A. Nash-Williams. Edge disjoint spanning trees of finite graphs. *J. London Math. Soc.*, 36:445–450, 1991.
22. J. Petersen. Die theorie der regulären graphs. *Acta Mathematica*, 15:193–220, 1891.
23. S. A. Plotkin, D. B. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Mathematics of Operations Research*, 20:257–301, 1995.
24. G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms*, 21(2):267–305, 1996.
25. N. Young. Randomized rounding without solving the linear program. In *Proc. 6th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 170–178, 1995.

# Coping with the NP-Hardness of the Graph Bandwidth Problem

Uriel Feige

Weizmann Institute, Rehovot 76100, Israel

**Abstract.** We review several approaches of coping with NP-hardness, and see how they apply (if at all) to the problem of computing the bandwidth of a graph.

## 1 Introduction

An important goal of the theory of algorithms is to produce efficient algorithms that solve computationally difficult problems. When considering the class of NP-hard combinatorial optimization problems, this goal is beyond our reach. As these problems need to be solved routinely, we seek ways of coping with NP-completeness. Perhaps the most common approaches are the following:

- **Easy special cases.** Do not solve the problem in its full generality. Identify properties of the input instances that make the problem easier, and design an algorithm that makes use of these properties.
- **Somewhat efficient algorithms.** Design an algorithm that always solves the problem whose running time is not polynomial, but still much faster than exhaustive search. This approach may be useful for inputs of moderate size.
- **Approximation algorithms.** Sacrifice the quality of the solution so as to obtain more efficient algorithms. Instead of finding the optimal solution, settle for a near optimal solution. Hopefully, this makes the problem easier.
- **Heuristics.** Design algorithms that work well on many instances, though not on all instances. This is perhaps the approach most commonly used in practice.

We review the four approaches mentioned above. For concreteness, we do so in the context of one particular NP-hard combinatorial optimization problem – that of computing a linear arrangement with minimum bandwidth for a graph. Given an  $n$ -vertex graph, a *linear arrangement* is a numbering of the vertices from 1 to  $n$  (which can be viewed as a layout of the graph vertices on a line) and its *bandwidth* is the maximum difference in numbers given to the endpoints of an edge (the maximum stretch of an edge on the line). The minimum bandwidth problem asks for a linear arrangement of minimum bandwidth. This problem is NP-hard [29].

## 2 Easy Special Cases

The bandwidth problem is a graph problem. By sufficiently restricting the class of graphs considered, one can identify classes of graphs for which the bandwidth can be computed in polynomial time (e.g., interval graphs [25]). Somewhat surprisingly, unlike many NP-complete problems, the bandwidth remains NP-hard on trees [17][28].

To see what kind of restriction on the input graph may be relevant to practical applications, let us consider a typical scenario in which the graph bandwidth problem arises. This scenario is that of minimizing the bandwidth of a matrix. For a symmetric matrix  $M$ , we say that its bandwidth is  $b$  if all its nonzero entries lie on entries at most  $b$  locations away from the diagonal. It is relatively easy to store and manipulate matrices of low bandwidth (e.g., diagonal matrices or tridiagonal matrices). Sometimes a symmetric matrix has large bandwidth, but can be transformed into a low bandwidth matrix just by renaming – applying a permutation on its rows, and the same permutation on its columns (to preserve symmetry). Finding a transformation that minimizes the bandwidth is equivalent to finding a linear arrangement of minimum bandwidth for a graph whose adjacency matrix is derived from  $M$  by replacing every nonzero entry by 1.

Minimizing the bandwidth of a matrix is worth the trouble only if the resulting matrix has small bandwidth. Hence a class of graphs for which the bandwidth problem is especially interesting is that of graphs with small bandwidth. Hence we may wish to study the complexity of the bandwidth problem as a function of  $b$ , the minimum bandwidth of the input graph.

For the case  $b = 1$ , the graph has to be a collection of paths, and finding a linear arrangement of minimum bandwidth is trivial. For the case  $b = 2$ , Garey, Graham, Johnson and Knuth [17] design a linear time algorithm for finding a linear arrangement with this bandwidth. They ask whether the case  $b = 3$  is NP-hard or polynomially solvable. This was answered by Saxe [30] who designed an algorithm that finds a linear arrangement of bandwidth  $b$  (if one exists) in time roughly  $O(n^{b+1})$ . The algorithm uses dynamic programming and is sketched below.

Build a linear arrangement of bandwidth at most  $b$  by extending partial linear arrangements (that include the first  $j$  vertices in the linear arrangement) by one more vertex, in all possible consistent ways. Namely, add one more vertex, such that no edge so far has length more than  $b$ , and moreover, such that this vertex did not previously appear in the linear arrangement. This last condition is the tricky one – how do we know which vertices appeared in the prefix?

To solve the above problem, we make the following two observations:

1. A graph has bandwidth at most  $b$  iff each of its connected components has bandwidth at most  $b$ . Hence we may w.l.o.g. assume that the graph is connected. This is a simple observation, but essential to the success of the algorithm.

2. Any set of consecutive vertices in a linear arrangement of bandwidth  $b$  has at most  $2b$  neighbors in  $G$ .

From a prefix of a linear arrangement of length  $j$ , we shall record only:

- The last  $b$  vertices, which we call the “buffer” vertices.
- Which of their  $2b$  neighbors have not yet appeared in the linear arrangement. We call these the “dangling” vertices. (In the dynamic programming algorithm, maintain only those partial linear arrangements in which the number of neighbors of the buffer vertices is at most  $2b$ .)

The point is that this representation implicitly includes all vertices in the prefix – those are the vertices which after the removal of the dangling vertices still have a path to some buffer vertex (due to observation 1 above). The amount of information we need to record is  $n^b$  possibilities for the buffer vertices, and at most  $2^{2b}$  possibilities for the dangling vertices (by observation 2 above), giving a polynomial running time when  $b$  is fixed.

Even though the algorithm runs in polynomial time for every fixed value of  $b$ , its running time is not practical even for moderate values of  $b$ . The question then arises of whether an algorithm can be found with better dependence on  $b$ . For example, is there an algorithm for computing the bandwidth that runs in time  $O(n^c f(b))$ , where  $c$  is some fixed constant independent of  $b$ , and  $f$  is an arbitrary function that does not depend on  $n$ . Note that a positive answer would not contradict the NP-hardness of the bandwidth problem (e.g., we can set  $f(b) = 2^b$  and then the running time becomes exponential when the bandwidth is large). Questions of this type are studied by the theory of parameterized complexity, and problems having a running time of the above form are called *fixed parameter tractable*. See [11] and references therein. Using reductions between parameterized NP-hard problems, a hierarchy of problems is established, in which problems that are hard for a certain level of the hierarchy are not fixed parameter tractable unless all problems lying in levels below it are. The bandwidth problem is hard for every fixed level of the fixed parameter tractability hierarchy [6].

### 3 Somewhat Efficient Algorithms

NP-complete problems can be solved by exhaustive search. The running time for exhaustive search becomes forbiddingly large already for instances of small size. Sometimes, it is possible to design algorithms that are significantly faster than exhaustive search, though still not polynomial time. This makes the solution of somewhat larger size problems possible. This is often the case for problems in number theory such as factoring (these problems are often not NP-hard, but still considered untractable), where for several decades there has been gradual improvement in the running time. See [1], for example. These improvements are of great significance to cryptography (and result in the need to use larger numbers in cryptosystems such as RSA).

For NP-hard combinatorial optimization problems, the improvements in running times are less dramatic. For problems such as 3SAT and max-CLIQUE, the current best running times are still  $O(c^n)$ , for some  $1 < c < 2$  (rather than roughly  $2^n$  which is the running time of exhaustive search).

For the bandwidth problem, we can exhaustively try all possible linear arrangements and choose the one with the lowest bandwidth. This has running time roughly  $n! \simeq n^n$ . Can we get the running time down to  $c^n$  for some  $c > 0$  independent of  $n$ ?

By analogy with other problems, we may hope that the answer is positive. Problems such as min-TSP, min-CUTWIDTH, minsum-LINEAR ARRANGEMENT also require an ordering of the vertices while minimizing a certain objective function (length of tour, maximum number of edges that cross a cut, sum of edge lengths, respectively). They can all be solved in time roughly  $2^n$  using dynamic programming (one need only remember which vertices appeared in the prefix of a linear arrangement, but not in what order they appeared). However, the same dynamic programming approach does not work for the bandwidth problem.

Here we sketch an algorithm (from [15]) for the bandwidth that runs in time  $c^n$ . For simplicity in this presentation, we shall assume that both  $n$  and  $b$  are powers of 2, and that  $G$  is connected.

1. First phase (finds to which segment each vertex belongs).
  - (a) Partition the interval  $[1, n]$  into  $2n/b$  segments of length  $b/2$ .
  - (b) Place vertex  $v_1$  in one of the segments. There are  $2n/b$  possibilities here, and all of them will be tried out using exhaustive search.
  - (c) Iteratively, take a yet unplaced vertex  $v$  that has a neighbor that is already placed, and place  $v$  in a segment that is of distance at most two from each one of its placed neighbors' segments. There are at most 5 possible segments available to  $v$ . There are  $n - 1$  vertices to place using this procedure, and hence at most  $5^{n-1}$  possible placements. All of them will be tried out using exhaustive search.

At the end of the first phase we have at most roughly  $5^n$  arrangements of vertices into segments, such that at least one of them is correct. Assume now that the second phase is performed with a correct arrangement (and multiply the running time by  $5^n$ ).

2. Second phase (finds exact locations within segments):
  - (a) Keep only edges that connect vertices that are two segments away, as all other edges will have length at most  $b$  regardless of the internal placement of vertices within segments. The problem now decomposes to two independent subproblems: that of finding a linear arrangement for vertices within the even numbered segments, and that of finding a linear arrangement for vertices within the odd numbered segments.
  - (b) For each subproblem recursively divide segments in two (of size  $b/4$  at the first step of the recursion, and sizes decrease by a factor of two with each new level), guess for each vertex whether to place it in the left half or right half of its segment, and then again partition the subproblem

into two independent problems, one involving the left side and the other involving the right side. For a subproblem involving  $k$  vertices, there are  $k$  guesses to make, leading to  $2^k$  possibilities, all of which are tried out. Keep only those possibilities in which no edge connects subsegments that are at distance more than  $b$  apart.

The number of possibilities tried out in the second phase satisfies the recursion  $T(k) \leq 2^k + 2T(k/2)$  implying roughly  $2^n$  possibilities altogether. Hence the running time of the algorithm is at most  $5^n 2^n = 10^n$  (up to polynomial factors). The base of the exponent can be somewhat improved using more careful analysis.

We note that a running time of  $10^n$  improves over  $n!$  only for values of  $n$  for which neither of these running times is practical. A more dramatic improvement in the running time would be desirable. For example, is there for every  $\epsilon > 0$  an algorithm that computes the bandwidth in time  $O(2^{\epsilon n})$ . We call such running times *weakly exponential*. Similar questions are being asked for other problems, such as 3SAT, and the answer is unknown. There is initial work on systematic study of the issue of weakly exponential running time. For example, one would like to establish for a large class of problems that either all of them have weakly exponential running times, or none of them do. This calls for reductions between instances that are linear in terms of the resulting problem size (rather than polynomial, as in the case of reductions establishing NP-hardness). The computation time of the reductions can be allowed to be weakly exponential. See [23, 22] for some interesting work in this respect.

For the bandwidth, the original reduction showing its NP-hardness [29] starts with a 3CNF formula with  $n$  variables, and ends with a graph with  $n^c$  vertices, where  $c > 1$ . However, it is possible to design other polynomial time reductions from 3SAT to bandwidth in which the number of vertices in the resulting graph is  $O(n)$  [15], establishing that bandwidth does not have weakly exponential algorithms unless 3SAT does.

## 4 Approximation Algorithms

Due to the intractability of the bandwidth problem, one may be willing to settle for a polynomial time algorithm that finds a linear arrangement whose bandwidth is not optimal, but also not much larger than optimum. We say that an algorithm has approximation ratio  $\rho(n)$  if on an  $n$  node graph it produces a linear arrangement whose bandwidth is within a factor of at most  $\rho(n)$  from optimal.

A known lower bound on the bandwidth is obtained via the *local density* bound. Let  $N(v, d)$  be the set of vertices at distance at most  $d$  from  $v$ . Then the local density of a graph is  $D = \max_{v,d} [|N(v, d)|/2d]$ , and the optimal bandwidth  $b$  satisfies  $b \geq D$ .

The algorithm with best approximation ratio known for the bandwidth problem produces (with high probability) a linear arrangement with bandwidth

$O(D(\log n)^3 \sqrt{\log n \log \log n})$  [12]. The algorithm can be interpreted in a geometric fashion as follows. It first embeds the vertices of the graph in high dimensional Euclidean space while balancing two conflicting requirements: keeping the Euclidean distance between vertices not larger than their distance in the graph, and making the volumes of the convex hulls of subsets of vertices of logarithmic size as large as possible. It then projects the geometric embedding on a random line and outputs the vertices in order of appearance on this line. The algorithm is nearly practical in terms of its running time (only slightly superlinear). Despite having an approximation ratio that is by far superior to that of any other known algorithm, the algorithm fails to take advantage of easy input instances, and often produces linear arrangements of higher bandwidth than those produced by other algorithms. For trees, Gupta [19] shows an algorithm that produces a linear arrangement with bandwidth  $O(D(\log n)^{5/2})$ . The analysis of this algorithm borrows ideas from the analysis in [12], but the algorithm itself is different, and is more likely to produce good linear arrangements in practice (though only for trees).

The analysis of the known approximation algorithms for the bandwidth compares the bandwidth of the linear arrangement obtained to the local density of the graph. Such an approach is not likely to produce approximation algorithms with sublogarithmic approximation ratios (compared to the true bandwidth). There are families of graphs with local density bounded above by a universal constant, whereas their bandwidth can be arbitrarily large [9, 8]. A gap of  $\Omega(\log n)$  between local density and bandwidth can be demonstrated on trees [9] and on expander graphs.

It is questionable whether there are polynomial time algorithms with sublogarithmic approximation ratios for the bandwidth. Blache, Karpinski and Wirtgen [3] showed that it is NP-hard to approximate the bandwidth within a ratio better than  $3/2$ . This was later improved by Unger to every constant factor [32]. Presumably, Unger's result can be extended to showing that the bandwidth cannot be approximated within a ratio that is a slowly growing function of  $n$ , unless 3SAT has subexponential algorithms. It would be interesting to see whether this function comes close to  $\log n$ .

## 5 Heuristics

In practice, heuristics for minimizing the bandwidth appear to work rather well [10]. These heuristics are often based on numbering the vertices in breadth first search order, or on simple variations on this approach.

A theoretical explanation for the success of heuristics is given by Turner [31]. He studies a random graph model in which a random graph with edge probability  $p$  is forced to have bandwidth at most  $b$  by deleting all edges that connect vertices whose indices differ by more than  $b$ . Thereafter the names of vertices are permuted at random and the resulting graph is given as input to the heuristic. Turner shows that a heuristic similar to BFS recovers for most such graphs a

linear arrangement of bandwidth at most  $b + O(\log n)$ . This is almost optimal (when  $b \gg \log n$ ) as it can be shown that the bandwidth of most such graphs is at least  $b - O(\log n)$ .

Turner assumes in his work that  $0 < p < 1$  is some constant independent of  $n$ . The case where  $p$  may depend on  $n$  was handled in [16], where a modified algorithm is showed to produce a linear arrangement of bandwidth at most  $(1 + \epsilon)b$ , when  $p > \log n/b$ . This modified algorithm produces two linear arrangements (essentially performing BFS once from the left most vertex and once from the right most vertex) and combines them into one linear arrangement. It is also shown in [16] that the algorithm extends to a *semirandom* graph model, in which an adversary is allowed to add arbitrary edges of its choice to the random graph prior to the deletion of the long edges.

There are graphs on which the heuristics mentioned above output a linear arrangement with bandwidth that is a factor of  $\Omega(n/\log n)$  larger than optimal. However, the quality of a heuristic is measured neither with respect to worst case input instances, nor with respect to best case input instances. To evaluate a heuristic we need a notion of an average case input instance. This is a very elusive concept, and there is no one particular model for average case analysis. Unlike the case of approximation algorithms, in which we can order the quality of approximation algorithms by their worst case approximation ratio, there does not seem to be any agreed upon way of deciding which of two heuristics give better results (unless one of the heuristics is better than the other on *every* instance). There is also a great difficulty of establishing negative results for heuristics. We describe below two methodologies for establishing limitations on what can be achieved using heuristics. So far, neither of them had major impact on the theory of heuristics, and much work remains to be done in this respect.

Levin [27] (see also [18]) develops a theory of average case polynomial time. In his framework, we associate probability distributions with problems, and sample input instances using this probability distribution. Then an algorithm needs to solve the input problem in average polynomial time, where averaging takes into account the probability of generating the input instance. One may argue that a distributional problem is hard on average if every other NP-problem with a polynomially sampleable distribution can be reduced to this problem. Some distributional problems are known to be hard under this notion, and it remains a challenge to use this notion to prove average case hardness results for common NP-problems under “natural” distributions.

Another approach for proving hardness results is to work in a semirandom model in which the input instance is chosen at random and then modified (subject to some constraints) by an adversary. There it is sometimes possible to show that by varying some parameter of the input instances (that controls what fraction of the input is random and what fraction is adversarial) there is a shift from classes of inputs that are polynomial time solvable on average to classes of inputs that are NP-hard on average. See [5, 14] for more details.

## 6 Conclusions

We discussed four approaches for dealing with NP-hard problems. Three of these approaches (parameterized versions of the problem, weakly exponential time algorithms, approximation algorithms) involve worst case performance measures, leading to familiar methods for evaluating and comparing the quality of algorithms. Reductions between problems is a very important tool in this respect. The fourth approach (heuristics) involves average case performance measures, and conceptual work is still needed in defining measures for quantifying the quality of a heuristic.

Of course, one can study mixtures of the approaches presented above. For example, one can mix the first and third approach and study approximation algorithms for special families of graphs. Indeed, this was done for the bandwidth problem [21][24][26]. As another example, one may mix the first three approaches, and study relations between the approximability of parameterized versions of problems and the existence of weakly exponential time algorithms. In [13] it is shown that if one can distinguish in polynomial time between graphs with cliques of size at most  $\log n$  and graphs with cliques of size at least  $2 \log n$ , then 3SAT can be solved in expected time roughly  $2^{\sqrt{n}}$ .

Of the questions that remain open for the bandwidth problem, let us mention three:

1. Does the bandwidth problem have considerably faster exponential time algorithms? E.g., can it be solved in time roughly  $2^n$  (rather than  $10^n$ , as shown in [15])?
2. Does the local density approximate the bandwidth within a logarithmic factor (rather than polylogarithmic, as shown in [12])? That is, is it true that  $b = O(D \log n)$ .
3. For a random graph (with constant edge probability  $p$ ), remove all edges that connect vertices whose indices differ by more than  $b$ . The bandwidth of the resulting graph is at most  $b$ . Turner [31] shows that it is at least  $b - O(\log n)$  (under some restrictions on the size of  $b$ ). When  $b \ll \sqrt{n/\ln n}$ , the bandwidth is at least  $b - O(1)$  [16]. Is the bandwidth of these random graphs exactly  $b$  (with high probability)?

## Acknowledgements

The author is the Incumbent of the Joseph and Celia Reskin Career Development Chair. Part of this work is supported by a Minerva grant, project number 8354 at the Weizmann Institute.

## References

1. L. Adleman. "Algorithmic Number Theory – The Complexity Contribution". *Proc. of 35th FOCS*, 1994, 88–113.
2. S. Assman, G. Peck, M. Syslo, J. Zak. "The bandwidth of caterpillars with hairs of length 1 and 2". *SIAM J. Algebraic Discrete Methods* 2 (1981), 387–393.
3. G. Blache, M. Karpinski, J. Wirtgen. "On approximation intractability of the bandwidth problem". Manuscript, 1998.
4. A. Blum, G. Konjevod, R. Ravi, S. Vempala. "Semidefinite relaxations for minimum bandwidth and other vertex-ordering problems". *Proc. of 30th STOC*, 1998, 100–105.
5. A. Blum, J. Spencer. "Coloring Random and Semi-Random k-Colorable Graphs". *Journal of Algorithms* 19, 204–234, 1995.
6. H. Bodlaender, M. Fellows, M. Hallet. "Beyond NP-completeness for problems of bounded width: hardness for the W Hierarchy". *Proc. of 26th STOC*, 1994, 449–458.
7. P. Chinn, J. Chvatalova, A. Dewdney, N. Gibbs. "The bandwidth problem for graphs and matrices – a survey". *Journal of Graph Theory*, 6 (1982), 223–254.
8. F. Chung, P. Seymour. "Graphs with small bandwidth and cutwidth". *Discrete Mathematics* 75 (1989) 113–119.
9. J. Chvatalova. On the bandwidth problem for graphs, Ph.D. dissertation, University of Waterloo, 1980.
10. E. Cuthill, J. McKee. "Reducing the bandwidth of sparse symmetric matrices". *ACM National Conference Proceedings*, 24, 1969, 157–172.
11. R. Downey, M. Fellows. Parameterized Complexity. Springer-Verlag New York, 1999.
12. U. Feige. "Approximating the Bandwidth via Volume Respecting Embeddings". *Journal of Computer and System Sciences*, to appear. (A preliminary version appeared in the proceedings of the 30th STOC, 1998, 90–99.)
13. U. Feige, J. Kilian. "On limited versus polynomial nondeterminism". *Chicago Journal of Theoretical Computer Science*, 12 March 1997.  
<http://www.cs.uchicago.edu/pub/publications/cjtcs/index.html>
14. U. Feige, J. Kilian. "Heuristics for semirandom graph problems" *Journal of Computer and System Sciences*, to appear. (Preliminary version in *Proc. of 39th FOCS*, 1998, 674–683.)
15. U. Feige, J. Kilian. "Exponential time algorithms for computing the bandwidth of a graph". *Manuscript in preparation*.
16. U. Feige, R. Krauthgamer. "Improved performance guarantees for bandwidth minimization heuristics". *Manuscript*, 1998.
17. M. Garey, R. Graham, D. Johnson, D. Knuth. "Complexity results for bandwidth minimization". *SIAM J. Appl. Math.* 34 (1978), 477–495.
18. O. Goldreich. "Notes on Levin's Theory of Average-Case Complexity". *Manuscript*, 1997.  
[www.wisdom.weizmann.ac.il/home/odedg](http://www.wisdom.weizmann.ac.il/home/odedg)
19. A. Gupta. "Improved bandwidth approximation algorithms for trees". *Proc. SODA 2000*.
20. E. Gurari, I. Sudborough. "Improved dynamic programming algorithms for bandwidth minimization and the min-cut linear arrangement problems". *J. Algorithms*, 5 (1984), 531–546.

21. J. Haralambides, F. Makedon, B. Monien. “Bandwidth minimization: an approximation algorithm for caterpillars”. *Math Systems Theory* 24, 169–177 (1991).
22. R. Impagliazzo, R. Paturi. “Complexity of k-SAT”. *Proc. Computational Complexity*, 1999.
23. R. Impagliazzo, R. Paturi, F. Zane. “Which problems have strongly exponential complexity”. *Proc. FOCS 1988*, 653–663.
24. M. Karpinski, J. Wirtgen, A. Zelikovsky. “An approximation algorithm for the bandwidth problem on dense graphs”. *ECCC TR97-017*.
25. D. Kleitman, R. Vohra. “Computing the bandwidth of interval graphs”. *SIAM J. Discrete Math.*, 3 (1990), 373–375.
26. T. Kloks, D. Kratsch, H. Muller. “Approximating the bandwidth for asteroidal triple-free graphs”. In *Algorithms – ESA '95, Paul Spirakis (Ed.), Lecture Notes in Computer Science* 979, 434–447, Springer.
27. L. Levin. “Average case complete problems”. *SIAM J. Comput.*, 15(1):285–286, 1986.
28. B. Monien. “The bandwidth-minimization problem for caterpillars with hair-length 3 is NP-complete”. *SIAM J. Algebraic Discrete Methods* 7 (1986), 505–512.
29. C. Papadimitriou. “The NP-completeness of the bandwidth minimization problem”. *Computing* 16 (1976), 263–270.
30. J. Saxe. “Dynamic programming algorithms for recognizing small-bandwidth graphs in polynomial time”. *SIAM Journal on Algebraic Methods* 1 (1980), 363–369.
31. J. Turner. “On the probable performance of heuristics for bandwidth minimization”. *SIAM J. Comput.*, 15, (1986), 561–580.
32. W. Unger. “The complexity of the approximation of the bandwidth problem”. In *Proc. 39th Annual IEEE Symposium on Foundations of Computer Science*, 1998, 82–91.

# Toward Complete Genome Data Mining in Computational Biology \*

Esko Ukkonen

Department of Computer Science  
P.O.Box 26, FIN-00014 University of Helsinki, Finland  
[ukkonen@cs.Helsinki.Fi](mailto:ukkonen@cs.Helsinki.Fi)

**Summary.** The invention of the so-called DNA sequencing more than 20 years ago has by now created an exponentially exploding flood of sequence data. For a computer scientist, such data consists of strings of symbols in an alphabet of size four. Being discrete by nature, the analysis and handling of sequence data is an exceptionally attractive and – noting its role in the heart of life – challenging application domain for combinatorial algorithmics. Hence it does not come as a surprise that computational molecular biology and bioinformatics are currently very active interdisciplinary research areas [5,10].

The algorithms for solving the so-called DNA fragment assembly problem and their implementations used as an integral part of the DNA sequencing process are one of the major successes of computational biology. The early developments [8,7] have been followed by more sophisticated methods [3]. This line of research culminated on the recent announcement by Celera Genomics of completed sequencing of the entire human genome. Computational methods have had a crucial role in this achievement.

Another important success of computational biology is the creation of sequence databases and, in particular, the development of the very fast methods such as BLAST for homology searching, that is, for finding in the database the sequences that are approximately similar to a given sequence [2]. Such searches are routinely used in biological research to compare any new sequence against all old ones.

The availability of entire genomes of several organisms as well as some new measuring instruments such as the DNA microarrays are rapidly introducing new computational problems. Knowing the raw DNA sequence of a genome is just a starting point for more refined analysis. The DNA microarrays give time-series data on the expression levels of the genes, i.e., how actively each gene is used, during different phases of the development of the organism or under external stress or diseases [14]. Analysing this rich data together with the genomic sequence itself opens new opportunities to trace the 'run-time behaviour' of the 'program' encoded into the genome.

For example, the following data analysis scenario can be followed: First, find potentially co-regulated groups of genes by clustering together the genes with similar expression level profiles. Next, pick from the genome the so-called regulatory regions associated with each gene in each group. Then, search for patterns

---

\* A work supported by the Academy of Finland.

of symbols that are overrepresented in the regulatory regions for each group. Such patterns are potential transcription binding sites, i.e., sites in the genome where a protein, specific for the pattern, binds itself and regulates in this way the use of the gene.

I will discuss in the talk our work [9] along this line, applied to the yeast (*Saccharomyces cerevisiae*) genome. Different clustering problems are the most interesting algorithmic tasks contained in this type of study. One of them, namely finding common patterns of symbols in a set of sequences, will be discussed in more detail. A quite general and flexible solution can be obtained using simple suffix-tree techniques [6].

To conclude, it should be emphasized that in computational biology the typical data is noisy and incomplete. Hence the algorithms must be robust and noise-tolerant, both properties that are often ignored in theoretical algorithmics. Statistical considerations are also becoming more and more important. As a positive remark it should be noted that, perhaps unexpectedly, the genomes are not intolerably large. The speed and storage capacity of the basic PC is increasing rapidly. Therefore it will soon be possible to store and mine the entire 3 billion bases long human genome in the core memory of your desktop computer.

## References

1. A. Alizadeh *et al.*, Distinct types of diffuse large B-cell lymphoma identified by gene expression profiling, *Nature* 403 (3 February 2000) 503–511.
2. S. Altschul, W. Gish, W. Miller, E. W. Myers & D. Lipman, A basic local alignment search tool, *J. Mol. Biol* 215 (1990) 403–410.
3. E. Anson & G. Myers, Algorithms for whole genome shotgun sequencing, *RECOMB99*, ACM Press (1999) 1–9.
4. J. L. DeRisi, V. R. Iyer & P. O. Brown, Exploring the metabolic and genetic control of gene expression on a genomic scale, *Science* 278 (1997) 680–686.
5. D. Gusfield, *Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology*, Cambridge University Press 1997.
6. [http://www.ebi.ac.uk/~vilo/Expression\\_Profiler/](http://www.ebi.ac.uk/~vilo/Expression_Profiler/)
7. H. Peltola, H. Söderlund, J. Tarhio & E. Ukkonen, Algorithms for some string matching problems arising in molecular genetics, *Proc. 9th IFIP World Computer Congress*, Elsevier (1983) 59–64.
8. R. Staden, Automation of the computer handling of gel reading data produced by the shotgun method of DNA sequencing, *Nucleic Acids Research* 10 (1982), 4731–4751.
9. J. Vilo, A. Brazma, I. Jonassen, A. Robinson & E. Ukkonen, Mining for putative regulatory elements in the yeast genome using gene expression data, *Proc. Eighth Int. Conference on Intelligent Systems for Molecular Biology ISMB 2000*, in press, AAAI Press (2000).
10. M. S. Waterman, *Introduction to Computational Biology: Maps, Sequences, Genomes*, Chapman and Hall 1995.

# A New Trade-Off for Deterministic Dictionaries

Rasmus Pagh

BRICS\*, Department of Computer Science, University of Aarhus,  
8000 Aarhus C, Denmark  
[pagh@brics.dk](mailto:pagh@brics.dk)

**Abstract.** We consider dictionaries over the universe  $U = \{0, 1\}^w$  on a unit-cost RAM with word size  $w$  and a standard instruction set. We present a linear space deterministic dictionary with membership queries in time  $(\log \log n)^{O(1)}$  and updates in time  $(\log n)^{O(1)}$ , where  $n$  is the size of the set stored. This is the first such data structure to simultaneously achieve query time  $(\log n)^{o(1)}$  and update time  $O(2^{(\log n)^c})$  for a constant  $c < 1$ .

## 1 Introduction

Among the most fundamental data structures is the *dictionary*. A dictionary stores a subset  $S$  of a universe  $U$ , offering membership queries of the form “ $x \in S?$ ”. The result of a membership query is either ‘no’ or a piece of *satellite data* associated with  $x$ . Updates of the set are supported via insertion and deletion of single elements.

Several performance measures are of interest for dictionaries: The amount of space used, the time needed to answer queries, and the time needed to perform updates. The most efficient dictionaries known depend on a source of random bits (are randomized, as opposed to deterministic). However, being randomized means that either: 1. There is a chance that the expected time bounds do not hold, or 2. There is a chance of the data structure returning a wrong answer. In some situations, this may not be acceptable. Even if their use is acceptable, random bits may be expensive or unavailable. Finally, an understanding of the power of randomization is important from a theoretical point of view. All this has led to an interest in derandomization of known randomized algorithms and data structures. Several recent papers consider deterministic dictionaries [4, 10, 11, 12, 13]. However, previous space-efficient dictionaries with very fast lookups (time  $(\log n)^{o(1)}$ ) have had update time much larger than that of, say, binary search trees. Therefore these dictionaries are of interest mainly when insertions are quite rare compared to lookups. Our interest here lies in obtaining space-efficient deterministic dictionaries which combine fast updates (time  $(\log n)^{O(1)}$ ) with very fast lookups.

---

\* Basic Research in Computer Science,  
Centre of the Danish National Research Foundation.

The model of computation used is a unit-cost *word RAM*, in which each memory register contains a  $w$ -bit integer (a *word*). This model of computation, resembling modern computers, has been the object of much recent research. Hagerup's survey [9] contains a description of the model. We adopt the *multiplication model* whose instruction set includes addition, bitwise boolean operations, shifts and multiplication. Note that all operations can also be carried out in constant time on arguments spanning a constant number of words. The universe considered is the set of machine words,  $U = \{0, 1\}^w$ . For simplicity, we assume that each piece of satellite data occupies a single machine word (this could be a pointer to more bulky data). Throughout this paper,  $S$  will refer to a set of  $n$  elements from  $U$ . For notational convenience we omit the “time tag” on  $S$ ,  $n$  and other symbols denoting dynamically changing values. All bounds will be independent of  $w$ , unless explicitly stated. Note that the optimal space consumption of a dictionary is  $\Theta(n)$  words.

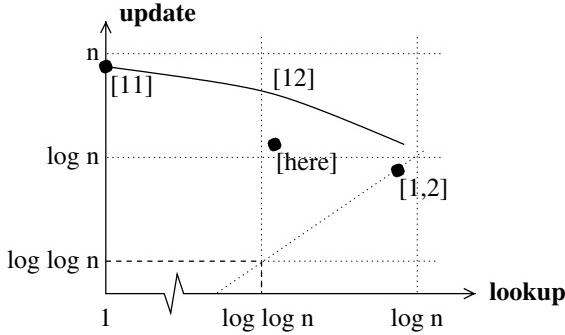
## 1.1 Related Work

The seminal result of Fredman, Komlós and Szemerédi [7] is that a *static* dictionary (i.e. without update operations) can have constant query time and linear space consumption. Allowing randomization, the FKS static dictionary can be made dynamic, supporting insertions and deletions in amortized expected constant time [4]. Improving this, Dietzfelbinger and Meyer auf der Heide [5] have constructed a dictionary in which all operations are done in constant time with high probability (i.e. probability at least  $1 - n^{-c}$ , where  $c$  is any constant of our choice). A simpler dictionary with the same properties was later developed [3]. As for randomized dictionaries, this leaves very little to be improved.

Without a source of random bits, the task of simultaneously achieving fast updates and constant query time seems considerably harder. The best deterministic dictionary with constant query time supports updates in time  $O(n^\epsilon)$ , for constant  $\epsilon > 0$  [11]. In fact, a range of trade-offs between update time and query time is known. For query time  $O(q(n))$ , where  $q(n) = O(\sqrt{\log n})$ , update time  $O(n^{1/q(n)})$  can be achieved [12]. The best known result in the situation where update and query time are considered equally important, is  $O(\sqrt{\log n / \log \log n})$  time per dictionary operation. It is a dynamization of the static data structure of Beame and Fich [2] using the exponential search trees of Andersson and Thorup [11].

The Beame-Fich-Andersson-Thorup (BFAT) data structure in fact supports *predecessor queries* of the form “What is the largest element of  $S$  not greater than  $x$ ?” Its time bound improves significantly if the word length is not too large compared to  $\log n$ . For example, if  $w = (\log n)^{O(1)}$ , the time per operation is  $O((\log \log n)^2 / \log \log \log n)$ . This will be a key component in our construction.

An unpublished manuscript by Sundar [13] states an amortized lower bound of time  $\Omega(\frac{\log \log_w n}{\log \log \log_w n})$  per operation for a deterministic dictionary in Yao's cell probe model [15], which in particular implies the same lower bound on the word RAM. Note that for  $w = (\log n)^{O(1)}$ , the BFAT data structure has time per



**Fig. 1.** Overview of deterministic dictionaries using linear space.

operation polynomially related to the lower bound. The challenge therefore seems to be finding ways of dealing with large word length.

## 1.2 This Work

In this paper we obtain a dictionary with query time  $O((\log \log n)^2 / \log \log \log n)$  and amortized update time  $O((\log n)^2)$  (we sketch how to make the latter bound worst-case). We deal with the problem of large word lengths by devising a dynamic *universe reduction* scheme, which reduces the problem to one within a smaller universe, which is then handled by the BFAT data structure. An interesting aspect of the reduction is that queries for the same element at two consecutive points in time usually translate to *different* BFAT queries. In particular, it is crucial that the BFAT data structure answers predecessor queries, and not just membership queries.

Our data structure is the first deterministic dictionary to simultaneously achieve query time  $(\log n)^{o(1)}$  and update time  $O(2^{(\log n)^c})$  for a constant  $c < 1$ . The data structure is *weakly non-uniform* in that it needs access to a fixed number of word-size constants depending (only) on  $w$ . These constants may be thought of as computed at “compile time”.

In the following we assume that  $w \geq (\log n)^5$ . Smaller word sizes can be handled using the BFAT data structure directly, and standard rebuilding techniques can be used to change from one data structure to the other. Similarly, we assume that  $n$  is larger than some fixed, sufficiently large constant, since constant size dictionaries are trivial to handle. We will look at machine words as binary numbers, with the most significant bits on the left and the least significant bits on the right. Bit positions are numbered from right to left, starting with zero.

## 2 Universe Reduction

Miltersen [41] has shown the utility of error-correcting codes to deterministic universe reduction. A *universe reduction function*  $\rho : U \rightarrow U'$  translates the

dictionary problem from universe  $U$  to the reduced universe  $U'$  (a search for  $x$  becomes a search for  $\rho(x)$ ). The advantage of this is that  $U'$  may be smaller and easier to handle. Previous universe reduction functions for the static dictionary problem [7, 11, 12] have been 1-1 on  $S$ . In the dynamic case this appears hard to combine with efficient updates, and in our construction the reduction function is  $O(\log n)$ -1. That is,  $O(\log n)$  elements of  $S$  may translate into the same element  $\rho(x)$ . A search among the elements “attached to  $\rho(x)$ ” is then needed to establish whether  $x \in S$ .

## 2.1 Error-Correcting Codes and Distinguishing Bits

Miltersen’s approach plays a key role in our construction, so we review it here. The basic idea is to employ an error-correcting code  $e : \{0, 1\}^w \rightarrow \{0, 1\}^{4w}$  and look at the dictionary problem for the transformed set  $\{e(x) \mid x \in S\}$ . For this it is possible to find a very simple function which is 1-1 on  $S$ , namely a projection onto  $O(\log n)$  bit positions.

The code must have relative minimum distance bounded from 0 by a fixed positive constant, that is, there must exist a constant  $\alpha > 0$  such that any two distinct codewords  $e(x)$  and  $e(y)$  have Hamming distance at least  $\alpha \cdot 4w$  (the supremum of such constants is called the relative minimum distance of the code). We can look at the transformed set without loss of generality, since Miltersen has shown that such an error-correcting code can be computed in constant time using multiplication:  $e(x) = c_w \cdot x$ , for suitable  $c_w \in \{0, 1\}^{3w}$ . The choice of  $c_w$  is a source of weak non-uniformity. The relative minimum distance for this code is greater than  $1/11$ . In the following,  $\alpha$  will denote a constant strictly smaller than the relative minimum distance of the error-correcting code (e.g.  $\alpha = 1/11$ ).

**Lemma 1.** (Miltersen) *For any  $R \subseteq U \times U$  there exists a discriminating bit position  $i \in \{0, \dots, 4w - 1\}$  such that  $|\{(x, y) \in R \mid x \neq y, e(x)_i = e(y)_i\}| \leq (1 - \alpha)|R|$ .*

**Corollary 2.** (Miltersen) *Let  $T$  be a set of  $m$  elements. There exists a set of distinguishing bit positions  $D \subseteq \{0, \dots, 4w - 1\}$  with  $|D| < \frac{2}{\alpha} \log m$  such that for all pairs of distinct elements  $x, y \in S$ , there is  $i \in D$  where  $e(x)_i \neq e(y)_i$ . The set  $D$  can be constructed deterministically in time  $O(m \log m)$ , given a deterministic  $O(m)$  time algorithm for finding a discriminating bit from the equivalence classes of an equivalence relation over  $T$ .*

*Proof sketch.* Elements of  $D$  may be found one by one, as discriminating bits of the equivalence relation where  $x, y \in T$  are equal iff  $e(x)$  and  $e(y)$  do not differ on the bit positions already chosen. The number of pairs not distinguished decreases exponentially with the number of bit positions chosen.  $\square$

Miltersen’s universe reduction function is simply  $x \mapsto e(x) \text{ AND } d$ , where AND denotes bitwise conjunction and  $d$  is the incidence vector of  $D$ . The reduced universe  $U'$  consists of the  $4w$ -bit vectors which are zero outside the positions given by  $D$ .

Two problems remain: 1. We must show how to find discriminating bit positions in time  $O(m)$ . 2. We want the reduction function to map to  $O(\log m)$  consecutive bits, that is, to  $\{0, 1\}^{O(\log m)}$ . The first problem was solved by Hagerup [10]. We need the following slight extension of his result to also solve the second problem:

**Lemma 3.** (Hagerup) *Given a set  $T$  of  $m$  elements, divided into equivalence classes, a discriminating bit position  $i$  can be found in time  $O(m)$  by a deterministic, weakly non-uniform algorithm. Further, for any set  $I \subseteq \{0, \dots, 4w-1\}$  of size  $O((\log n)^4)$  (given as a bit vector), we can assure that  $i \notin I$ .*

*Proof sketch.* It can be shown how to compute  $|\{\{x, y\} \subseteq T \mid x \neq y, x \equiv y, e(x)_i = e(y)_i\}|$  for all  $i \in \{0, \dots, 4w-1\}$  in time  $O(m)$ . The algorithm employs word-level parallelism, and the result vector spans  $O(\log m)$  words, since each number occupies  $O(\log m)$  bits. Word-parallel binary search can be used to find the smallest entry. To avoid entries in  $I$ , simply overwrite the entries of  $I$  with the largest possible integer before finding the minimum. This corresponds to changing the error-correcting code to be constant (i.e. non-discriminating) on the bit positions of  $I$ . Since  $|I| = O((\log n)^4)$  and the length of codewords is  $4w \geq 4(\log n)^5$ , the relative minimum distance of this modified code is still  $> \alpha$  (for  $n$  large enough). Hence, this procedure will find a discriminating bit position.  $\square$

## 2.2 Multiple Set Universe Reduction

To accommodate efficient updates, we will not maintain a set of distinguishing bit positions for  $S$  itself. Instead, we maintain  $k = \lceil \log(n+1) \rceil$  sets of distinguishing bit positions  $D_0, \dots, D_{k-1}$  for subsets  $S_0, \dots, S_{k-1}$  whose (disjoint) union is  $S$  and where  $|S_i| \in \{0, 2^i\}$ . By the results of Sect. 2.1 we can achieve  $|D_i| = O(i)$ , and recomputation of  $D_i$  when  $S_i$  changes takes time  $O(2^i i)$ . Additionally, we can make the complete set of distinguishing bit positions *well separated*, that is, no pair of positions differ by less than  $2c(\log n)^2$ , where  $c$  is a suitably large constant.

Since the distinguishing bit positions are well separated, we are able to “collect” and order the distinguishing bits within  $O((\log n)^2)$  consecutive bit positions, such that the distinguishing bits of  $S_0$  are least significant, and the distinguishing bits of  $S_{k-1}$  are most significant. For each empty set  $S_i$  we will have a number of zero-bits. The following lemma makes this precise.

**Lemma 4.** *Given a list  $d_1, \dots, d_p$  of well separated bit positions, where  $p \leq c(\log n)^2$ , there is a function  $f_{\bar{d}} : \{0, 1\}^{4w} \mapsto \{0, 1\}^p$  such that for any  $x$ ,  $f_{\bar{d}}(x)_i = x_{d_i}$ . The function can be evaluated in constant time, and updated under changes of bit positions in constant time.*

*Proof.* We will show how to “move” bit  $d_i$  of  $x \in \{0, 1\}^{4w}$  to bit  $u+i$  of a  $u+p$ -bit string, where  $u \geq \max_i d_i$  (the desired value can then be obtained by shifting the word by  $u$  bits). We simply multiply  $x$  by  $m_{\bar{d}} = \sum_i 2^{u+i-d_i}$  (a method adopted

from [8, p. 428-429]). One can think of the multiplication as  $p$  shifted versions of  $x$  being added. Note that if there are no carries in this addition, we do indeed get the right bits moved to  $u+1, \dots, u+p+1$ . However, since the bit positions are well separated, all carries occur either left of the  $u+p$ th position (which is harmless) or right of position  $u-p$  (which can never influence the values at positions greater than  $u$ , since there are more than enough zeros in between to swallow all carries). Note that  $m_{\bar{d}}$  can be updated in constant time when a bit position changes.  $\square$

We are now ready to describe how to update the dynamic universe reduction function under updates. New elements are inserted in the lowest numbered empty set  $S_i$  together with the elements of  $S_0, \dots, S_{i-1}$  (these sets are then “emptied”). Note that the work per element when constructing a new set of distinguishing positions is  $O(\log n)$ . Since elements are always transferred to higher numbered sets, the total amortized work for an insertion is  $O(k \log n) = O((\log n)^2)$ . As we will see in the next section, this cost will be dominant in the cost of an insertion in the final dictionary.

The universe reduction function will not be updated during deletions. Rather, deletions are implemented by simply *marking* deleted elements in the dictionary. When more than half of the elements in the dictionary are marked, a new dictionary containing the unmarked elements is constructed. The cost of this is amortized over the deletions, which hence also have cost  $O((\log n)^2)$ .

### 3 Using the Predecessor Data Structure

Recall that our universe reduction function, which we will call  $\rho$ , computes the concatenation of functions  $f_k, \dots, f_0$  which are 1-1 on  $S_k, \dots, S_0$ , respectively. The value  $\rho(x)$  after  $x$  is inserted in  $S_i$  is used as key for  $x$  in the BFAT predecessor data structure. Functions  $f_0, \dots, f_{i-1}$  return zero vectors at this time. However, these functions will change in the period until the next update of  $S_i$ , and specifically  $f_0(x), \dots, f_{i-1}(x)$  may change. When a search for  $\rho(x)$  is conducted, the result will be either the BFAT key for  $x$ , or that of a key  $y$  later inserted, whose BFAT key agrees with that of  $x$  except possibly for some of the values of  $f_0, \dots, f_{i-1}$ . In this case we want  $x$  to be present in  $y$ 's associated (sorted) list of elements. That is, for each new key  $\rho(y)$  in the BFAT data structure, we want a list of elements which includes  $x \in S_i$  iff  $x$  and  $y$  agree on  $f_k, \dots, f_i$ .

A predecessor query on  $\rho(y) - 1$  will return the BFAT key which has the longest common prefix with  $y$  (if any). By invariant, the associated list of this key contains all the elements needed, apart from  $y$  itself, so it is easy to create the list associated with  $y$ . The crux is that, since  $f_k, \dots, f_0$  are 1-1, an associated list can contain at most one element from each set.

**Example.** We go through Fig. 2. This example has 3, 4 and 5 distinguishing bit positions for  $S_0$ ,  $S_1$  and  $S_2$ , respectively. The keys inserted in the BFAT

data structure are annotated with their list of elements. At  $t = 4$  the dictionary contains four elements, denoted  $a, b, c, d$ , all residing in  $S_2$ . At  $t = 5$  element  $e$  is inserted and put into  $S_0$ . The key for  $e$  coincides with the key for  $c$  on the first five bits, so the associated list contains  $c$  and  $e$ . A search for the key of  $c$  at this time would in fact find 00111 0000 000, so  $c$  is not strictly necessary in the new list. However, at  $t = 6$  element  $f$  enters, and  $S_1$  is filled by  $e$  and  $f$ . After this, a search for the key of  $c$  will find 00111 0010 000, and  $c$  can be found in the new list. At  $t = 7$  element  $g$  is inserted, and its key coincides with both the first five bits of  $c$ 's key and the first nine bits of  $e$ 's key, so the associated list becomes  $ceg$ .

	$t=4$	$t=5$	$t=6$	$t=7$
$S_2$	00010 0000 000 a	00010 0000 100	00010 0001 000	00010 0001 010
	00110 0000 000 b	00110 0000 110	00110 1000 000	00110 1000 000
	00111 0000 000 c	00111 0000 001	00111 1100 000	00111 0101 011
	11011 0000 000 d	11011 0000 110	11011 0010 000	11011 0010 001
$S_1$			00111 0010 000 ce	00111 0010 101
			11111 1101 000 f	11111 1101 100
$S_0$		00111 0000 011 ce		10111 0010 011 ceg

**Fig. 2.** Universe reduction function values for elements in  $S$  during three insertions.

### 3.1 Time and Space

A search for  $x$  requires computation of  $\rho(x)$  in constant time, a predecessor lookup in time  $O((\log \log n)^2 / \log \log \log n)$  and finally search of an associated list in time  $O(\log \log n)$ . That is, the total time is  $O((\log \log n)^2 / \log \log \log n)$ .

As for insertions, we already argued that the amortized cost of maintaining the universe reduction function is  $O((\log n)^2)$ , so we only need to see that the cost of maintaining the associated lists is no larger. This is not hard, since all that is needed is a single predecessor query and insertion of an element in a sorted list of length  $O(\log n)$ .

The only part of the data structure which is not clearly in linear space is the set of associated lists, where elements may occur  $\log n$  times. To see that their total length is  $O(n)$ , note that there can be no more than  $n/2^{i-1}$  lists of length  $i$ , since such lists must have been created in connection with insertion of elements in  $S_0, \dots, S_{k+1-i}$ .

## 4 Final Remarks

### 4.1 Speedups

Updates can be sped up slightly, to time  $O((\log n)^2 / \log \log n)$ , by using another strategy, in which there are  $\Theta(\log n)$  sets of each size, and only  $O(\log n / \log \log n)$  different set sizes. If the requirement of linear space is abandoned, substituting van Emde Boas trees [14] for the BFAT data structure gives membership queries in time  $O(\log \log n)$ . The space usage then rises to  $n^{O(\log n)}$ .

It can be noted that the predecessor data structure is used in such a way that it essentially answers “longest common prefix” queries on strings of length  $k+1$ , where the characters are described by the bits corresponding to sets  $S_k, \dots, S_0$ , respectively. A plausible way of improving the query time to, say,  $O(\log \log n)$  is by designing a faster data structure which can find such longest common prefixes.

### 4.2 Worst-Case Bounds

We gave amortized bounds. The same worst-case bounds follow by standard lazy rebuilding techniques, to be sketched below. Where the amortized insertion algorithm would “build”  $S_i$  and empty  $S_{i-1}, \dots, S_0$ , the *worst-case insertion* algorithm keeps  $S_{i-1}, \dots, S_0$  in memory and starts building  $S_i$  at a pace of  $c \log n$  steps per insertion (for some sufficiently large constant  $c$ ). Only when  $S_i$  is completed, we throw out the lower numbered sets.

More precisely, we now have sets  $S_{i,j}$  for  $0 \leq j < i \leq k$ , where  $|S_{i,j}| \in \{0, 2^j\}$ . The first index signifies that  $S_{i,j}$  will next become part of a new set of size  $2^i$ . Consider insertion number  $2^b d - 2^a$ , where  $a < b$  (any positive integer can be written like this for unique integers  $a$ ,  $b$  and  $d$ ). At this point we start constructing  $S_{b,a}$  from the new element and  $S_{a,0}, \dots, S_{a,a-1}$ . As the last stage of the construction, we set  $S_{a,0} = \dots = S_{a,a-1} = \emptyset$ . Constant  $c$  above can be chosen such that this is guaranteed to be finished before any of the sets  $S_{a,0}, \dots, S_{a,a-1}$  are to be reconstructed. The ordering of distinguishing bits is with respect to primarily the first index, secondarily the second index.

Since we need associated element lists of length  $\Omega((\log n)^2)$ , we cannot afford to use sorted lists as before (updates would become more expensive). Instead, we use persistent balanced search trees [6], which support updates and queries in time  $O(\log t)$  for a sequence of trees of size at most  $t$ . One technicality is that many instances of the algorithm finding distinguishing bits have to run at the same time and must produce well separated bit positions. However, since positions are chosen one by one, this poses no problem. In addition to what is done in the amortized case, the *worst-case deletion* algorithm inserts two elements of  $S$  in a new dictionary. When the transfer of all elements in  $S$  is completed, the new dictionary takes the place of the old one. Of course, transferred elements may be deleted before the new dictionary takes over.

## 5 Conclusion

We have seen a new lookup time vs insertion time trade-off for linear space deterministic dictionaries. This presents progress towards closing the gap between known upper and lower bounds. It also shows that universe reduction techniques have a place not only in the static setting.

The big open question is whether updates in such a dictionary can be accommodated in time  $(\log n)^{o(1)}$ . For example, time  $(\log \log n)^{O(1)}$  would mean that Sundar's lower bound is tight up to a polynomial. For  $w = (\log n)^{O(1)}$  this is achieved by the BFAT data structure. Thus, large word length seems to be the main enemy, and new universe reduction schemes with faster updates appear a promising approach.

## Acknowledgments

The author would like to thank Rolf Fagerberg and Jakob Pagter for useful feedback.

## References

- [1] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32th Annual ACM Symposium on Theory of Computing (STOC 2000)*, New York, 2000. ACM Press.
- [2] Paul Beame and Faith Fich. Optimal bounds for the predecessor problem. In *Proceedings of the 31th Annual ACM Symposium on Theory of Computing (STOC '99)*, pages 295–304, New York, 1999. ACM Press.
- [3] Martin Dietzfelbinger, Joseph Gil, Yossi Matias, and Nicholas Pippenger. Polynomial hash functions are reliable (extended abstract). In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246, Berlin, 1992. Springer-Verlag.
- [4] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, August 1994.
- [5] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Automata, languages and programming (Coventry, 1990)*, pages 6–19. Springer, New York, 1990.
- [6] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Comput. System Sci.*, 38(1):86–124, 1989. 18th Annual ACM Symposium on Theory of Computing (Berkeley, CA, 1986).
- [7] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with  $O(1)$  worst case access time. *J. Assoc. Comput. Mach.*, 31(3):538–544, 1984.
- [8] Michael L. Fredman and Dan E. Willard. Surpassing the information-theoretic bound with fusion trees. *J. Comput. System Sci.*, 47:424–436, 1993.
- [9] Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science (STACS '98)*, pages 366–398. Springer, Berlin, 1998.

- [10] Torben Hagerup. Fast deterministic construction of static dictionaries. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1999)*, pages 414–418, New York, 1999. ACM.
- [11] Peter Bro Miltersen. Error correcting codes, perfect hashing circuits, and deterministic dynamic dictionaries. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1998)*, pages 556–563, New York, 1998. ACM.
- [12] Rasmus Pagh. Faster deterministic dictionaries. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 487–493, New York, 2000. ACM.
- [13] R. Sundar. A lower bound on the cell probe complexity of the dictionary problem. Manuscript, 1993.
- [14] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Annual Symposium on Foundations of Computer Science (Berkeley, Calif., 1975)*, pages 75–84, Long Beach, Calif., 1975. IEEE Computer Society.
- [15] Andrew Chi Chih Yao. Should tables be sorted? *J. Assoc. Comput. Mach.*, 28(3):615–628, 1981.

# Improved Upper Bounds for Pairing Heaps

John Iacono\*

Department of Computer Science, Rutgers University, New Brunswick NJ 08903  
iacono@john.rutgers.edu

**Abstract.** Pairing heaps are shown to have constant amortized time insert and zero amortized time meld, thus improving the previous  $O(\log n)$  amortized time bound on these operations. It is also shown that pairing heaps have a distribution sensitive behavior whereby the cost to perform an extract-min on an element  $x$  is  $O(\log \min(n, k))$  where  $k$  is the number of heap operations performed since  $x$ 's insertion. Fredman has observed that pairing heaps can be used to merge sorted lists of varying sized optimally, within constant factors. Utilizing the distribution sensitive behavior of pairing heap, an alternative method the employs pairing heaps for optimal list merging is derived.

## 1 Introduction

Self adjusting data structures, through the use of simple update rules, are often able to match the asymptotic performance of non-self adjusting data structures over any sequence of operations. They do not store balance information and thus require less memory. Self adjusting structures are relatively easy to code and often empirically outperform their non-self adjusting counterparts. Some self adjusting data structures asymptotically perform as well as off-line algorithms on classes of execution sequences defined by various structural or distributional characteristics. Splay trees [9], a self adjusting binary search tree, have all of these qualities, and are clearly favorable over their non-self adjusting counterparts, both theoretically and empirically, in many situations. However, with respect to heap design, the self adjusting methodology has not achieved corresponding success. For pairing heaps, one form of self adjusting heap, we partially rectify this by asymptotically improving existing upper bounds to bring them closer to the best non-self adjusting data structure as well as introducing distribution sensitive upper bounds.

The leading non-self adjusting heap is the Fibonacci heap [7] which has constant amortized time *make-heap*, *insert*, *find-min*, *meld*, and *decrease-key*, while supporting *delete* and *extract-min* in  $O(\log n)$  amortized time. A recent alternative to Fibonacci heaps due to Kaplan and Tarjan, thin heaps [8], lowers the pointer and balance requirements, but remains cumbersome. It was conjectured in [6] and empirical evidence was presented in Stasko and Vitter [11] that pairing heaps share the same amortized cost per operation as Fibonacci heaps.

---

\* Research supported by NSF grant CCR-9732689.

However, this possibility was eliminated when it was shown by Fredman [4] that the amortized cost of *decrease-key* can not be below  $O(\log \log n)$ . In this work we present a tighter analysis of pairing heaps than found in [6] that proves, with the exception of *decrease-key* operations, pairing heaps share the same asymptotic runtime per operation as Fibonacci heaps. Specifically, the amortized upper bound of  $O(\log n)$  for the *insert* and *meld* operations, is improved to  $O(1)$  for *insert* and  $O(0)$ <sup>1</sup> for *meld*. It should be noted that Stasko and Vitter in [11] introduced a variant of pairing heaps, the auxiliary twopass method, and proved that this structure supported constant time *insert*. However, their analysis explicitly forbade the *decrease-key* operation.

Pairing heaps use a restructuring heuristic that bears a strong similarity to that of splay trees. While the ability of splay trees to exhibit certain types of distribution sensitive optimality has been extensively studied [9, 12, 2, 11, 13], such behavior, while expected in pairing heaps, has never been demonstrated. We prove one result, similar to the working set theorem for splay trees [9], that implies that if an item is in a pairing heap of maximum size  $n$  and  $k$  heap operations have been performed since its insertion, *extract-min* operations take amortized time  $O(\log \min(n, k))$ . This result holds for several variants of pairing heap and through the depletion transformation of Fredman [5], for top down skew heaps [10] as well. Our results are more robust than some of the results on splay trees, as we allow the heap size and contents to dynamically change, as opposed to the analyses of splay trees which only study the *access* operation when investigating distribution sensitive effects.

Fredman [3] has shown that  $n$  sorted lists of varying sizes can be optimally merged (within constant factors) using pairing heaps in the following manner: First, each sorted list is represented as a linked listed, viewed as a linearly structured heap-ordered tree. These trees are then combined into a single tree using  $n$  pairing heap *meld* operations. Finally, a single sorted list is obtained by executing repeated *extract-min* operations. Inspired by Fredman's results, an alternative approach to list merging proceeds by inserting the smallest element from each list into a pairing heap, and then repeatedly executing the pattern: *extract-min*, *insert*; where each insertion involves the next element from the input list that contains the previously extracted element. An application of the  $O(\log \min(n, k))$  result shows that this approach to list merging achieves optimal performance, within constant factors, in both pairing and top-down skew heaps.

## 2 Pairing Heaps

A pairing heap is a heap ordered general tree. The basic operation on a pairing heap is the pairing operation, which combines two pairing heaps by attaching the root with the larger key value to the other root as its leftmost child. Priority

---

<sup>1</sup> *Meld* in Fibonacci heaps is typically stated as taking  $O(1)$  amortized time. However, since *meld* operations must be dominated by *make-heap* operations, *meld* operations can never asymptotically change the runtime of any sequence, and thus take  $O(0)$  amortized time.

queue operations are implemented in a pairing heap as follows: *Make-heap* creates a new single node heap. *Find-min* returns the data in the root of the heap. *Merge* pairs the roots of the two heaps. *Insert* pairs the new node with the root of heap. *Decrease-key* breaks off the node and its subtree from the heap (if the node is not the root), decreases the key value, and then pairs it with the root of the heap. *Delete* breaks off the node to be deleted and its subtree, performs an *extract-min* on the subtree, and pairs the resultant tree to the root of the heap. *Extract-min* removes and returns the root, and then, in pairs, pairs the remaining trees. Then, the remaining trees from right to left are incrementally paired. All pairing heap operations take constant actual time, except *extract-min* and *delete*, which take time linear in the number of children of the node to be removed. For the purposes of implementation, pairing heaps are stored as a binary tree using the leftmost child, right sibling correspondence. Unless otherwise stated, the standard tree terminology will refer to the general tree representation.

### 3 Constant Amortized Time Insert and Zero Amortized Time Meld in Pairing Heaps

We claim that in a pairing heap the amortized runtime of *find-min*, *make-heap*, and *insert* is  $O(1)$ , *meld* is  $O(0)$  and *decrease-key*, *delete* and *extract-min* is  $O(\log n)$ . The  $n$  used in the analysis is number of items in the heap that will be removed during execution sequence in question, rather than simply the total number of items in the heap. Proving these amortized costs is equivalent to proving the following statement:

**Theorem** Given a sequence  $S = s_1 \dots s_m$  of  $m$  operations, where  $D = \{i | s_i \text{ is a } \textit{extract-min}, \textit{decrease-key} \text{ or } \textit{delete} \text{ operation}\}$ ,  $C = \{i | s_i \text{ is a } \textit{find-min}, \textit{make-heap}, \text{ or } \textit{insert} \text{ operation}\}$ , and  $n_i$  is the size of the heap before the execution of  $s_i$ ,  $S$  can be executed on an initially empty forest of paring heaps in time  $O(|C| + \sum_{i \in D} \log n_i)$ . Note that *meld* operations are allowed but can never asymptotically affect the runtime of any sequence; that is the meaning of  $O(0)$  amortized time.

**Proof** The potential method is used. The amortized time of operation  $i$ ,  $\hat{a}_i$ , is defined to be the actual time of the operation,  $a_i$ , plus the change in potential,  $\Phi_i - \Phi_{i-1}$ . Summing over the sequence  $S$  and rearranging yields:  $\sum_{i=1}^m a_i = \sum_{i=1}^m \hat{a}_i + \Phi_0 - \Phi_m$ . Thus, the actual runtime of a sequence of operations is equal to the sum of the amortized time of the operations plus the net loss of potential. Note that the amortized times that we prove below are different then the ones stated above; We prove these amortized times to bound the runtime of the entire sequence, and that in turn proves our originally stated amortized costs.

For the analysis, a color, black or white, is assigned to every node, and a weight is assigned to those nodes colored white. A node is black if it will remain in the forest of heaps at the end of execution of sequence  $S$ , and white otherwise. We say that a white node is heavy if the number of white nodes in its left subtree in the binary representation is greater than or equal to the number of white

nodes in its right subtree. Roots and leaves are always heavy by this definition and every node can have a maximum of  $\log n$  heavy children. We say that a white node that is not heavy is light.

We say a white node has been captured if its parent is black. Captured nodes must have either a *decrease-key* or a *delete* performed on them later in the execution sequence, and until such time they are not involved in any pairings.

The potential of a white node is the sum of four components: Rank potential, weight potential, triple white potential, and capture potential. The rank potential of a white node is the 9 times the logarithm of the number of white nodes in its induced subtree in the binary representation. If node is white and has right and left siblings that are also white, then the node has a triple white potential of  $-6$ , else it has no triple white potential. Heavy nodes have a weight potential of  $-6$ , and light nodes have no weight potential. We assign captured nodes a capture potential of  $-6$  and non-captured nodes no capture potential. The potential of a black node is  $-6$  if its parent, in the general representation, is black, and 0 otherwise. The potential of a forest of heaps,  $\Phi$ , is the sum of the potentials of the nodes in the heaps.

The amortized cost of each operation is now calculated using this potential function:

Amortized cost of *create-heap* is  $O(1)$ : Actual cost is 1, and the newly inserted node has at most 0 potential (This is true if it is white or black). Thus the amortized cost is at most 1.

Amortized cost of *Insert* is  $O(\log n)$ , if the newly inserted node is white: Actual cost is 1. The newly inserted node will have a potential at most  $9 \log n$ . The old root is the only other node that will change potential, gaining at most  $9 \log n$ . Thus the amortized cost of this operation, which is the sum of the actual cost and the change in potential, is  $1 + 18 \log n$ .

Amortized cost of *Insert* is  $O(1)$ , if the newly inserted node is black: Actual cost is 1, and there are no gains of potential, for an amortized cost of 1.

Amortized cost of *Meld* on two trees with black roots, or on a white tree with a white root, and the root of a heap composed entirely of black nodes is  $O(1)$ : Actual cost is 1, and there are no potential increases. Thus the amortized cost is 1.

Amortized cost of *Meld* on two heaps with white roots or one heap with a white root and one heap with a black node that is the root of a heap that contains at least one captured white node is  $O(\log n)$ : Actual cost is 1. At most two nodes, the roots of the trees to be melded, increase potential. Both could gain rank potential, up to  $9 \log n$  each. Also the root with smaller key value could change from heavy to light, causing a gain of 6. Thus the amortized cost is  $7 + 18 \log n$

Amortized cost of *decrease-key* is  $O(\log n)$  Actual cost is 1. The node on which the *decrease-key* is performed could gain as much as  $9 \log n$  in rank potential, 6 in weight potential, and 6 in capture potential. Among the node on which the *decrease-key* is performed, and its two former siblings to the left and right, a total of 6 units of triple white potential can be gained. Also, on the path from

the node on which the *decrease-key* is to be performed to the root, the removal of the node and its subtree may cause some nodes to change their status from light to heavy or vice versa. Only changing from heavy to light causes a potential gain, and this gain of 6 can only happen in  $\log n$  nodes. Thus the amortized cost is  $19 + 15 \log n$ .

Amortized cost of *extract-min* is  $O(\log n)$ :

If there are  $c$  children of the root, the actual cost is  $c - 1$ . The removal of the root itself causes a potential gain of at most 6, since it was heavy, and is not captured or a triple-white. There are at most  $\log n$  heavy children of the root, and so the potential gain caused by heavy nodes becoming light is at most  $6 \log n$ . Given that there are  $w$  white-white pairings in the first pairing pass, the first pairing pass causes a rank potential gain of at most  $18 \log n - 18w$ . The second pairing pass causes a rank potential gain of at most  $9 \log n$ . The derivation of the changes in rank potential may be found in [6].

The *extract-min* operation can cause no increase in capture potential, as none of the children of the are be captured.

In order to analyze other changes in potential (changes in triple white potential, losses of weight potential caused by a node becoming heavy, and changes in black nodes' potential) we break the children of the root into blocks of six nodes, excluding the rightmost two nodes. At most 7 nodes can not be included in this analysis, and they could incur a potential gain of up to 12 each. In analyzing these specific potential changes in each block of six, there are six cases.

Case 1: There is at least one white-white pairing in the first pairing pass.

The only gains in potential are the possible gain of 6 units of triple white potential for each white involved in a white-white pairing. This is the only case where a gain in potential, among the components of the potential function under consideration is possible.

Case 2: There are no white-white pairings and at least one black-black pairing in the first pairing pass.

The black-black pairing(s) causes a loss of at least 6 units of potential.

Case 3: All are black-white pairings in the first pairing pass, and at least one of the three white nodes is captured.

The capturing of the node(s) causes a capture potential loss of at least 6.

Case 4: All are black-white pairings in the first pairing pass, but all three nodes that participate in the second pairing pass lose.

Having all three loose the pairings in the second pairing pass causes a loss of potential of 6, due to the change of status of the middle white node to a triple white.

Case 5: All are black-white pairings in the first pairing pass, and at least one of the three nodes that participate in the second pairing pass wins, and at least one of the white nodes is light.

The light node becomes heavy, as all nodes previously on its right are now in its subtree. Additional nodes that were to the node's left may also be added to its subtree, but this just makes it more heavy. This causes a loss of 6 units of heavy potential.

Case 6: All are black-white pairings in the first pairing pass, and at least one of the nodes that participates in the second pairing pass wins, and all of the whites that win in the second pairing pass are heavy.

There is no potential gain. This case can only happen  $\log n$  times, because there are at most  $\log n$  heavy children of the root.

Case 1 causes a potential gain of at most  $12w$ , and cases 2-6 cause a potential loss of at least  $-6(\lfloor \frac{c-2}{6} \rfloor - w - \log n)$ . These potential changes are in addition to the gain of at most  $33 \log n - 18w$  discussed earlier. Thus summing the actual cost of the *extract-min* operation,  $c - 1$  with the maximum potential gain yields an amortized cost of  $89 + 39 \log n$ .

The amortized cost of *delete* is  $O(\log n)$ : If the node to be deleted has  $c$  children, the actual cost to delete a node is  $c - 1$ . The removal of the node and its subtree can cause a weight potential gain of  $6 \log n$  in its ancestors. Among the node on which the *delete* is performed, and its two former siblings to the left and right, a total of 6 units of triple white potential can be gained. Performing an *extract-min* on the newly extracted subtree causes a potential gain of at most  $89 + 39 \log n - c$  (See analysis of *extract-min* above). Pairing the resultant tree with the root of the original tree causes a potential gain of at most  $6 + 18 \log n$  (see analysis of *meld*). Thus the total potential gain is at most  $101 + 63 \log n - c$  and the amortized cost is  $100 + 63 \log n$

The total potential loss over the execution of the sequence  $S$ ,  $\Phi_0 - \Phi_m$ , is at most  $-6|C|$ : The initial potential of the empty data structure is 0 potential is zero. At the end of the execution sequence, the data structure is a forest of black nodes. Each non-root node has a potential of  $-6$ . Since  $|C|$  is at least the total number of nodes inserted into the structure, the total potential loss is at most  $-6|C|$ .

The sum of the amortized cost to perform the *extract-min*, *decrease-key* and *delete* operations is  $O(\sum_{i \in D} \log n_i)$ . It can also been seen that the sum of the  $O(\log n)$  amortized costs to *insert* white nodes and to perform *meld* operations on two heaps where the root of one of the heaps is white and the other heap contains at least one white node can not exceed  $O(\sum_{i \in D} \log n_i)$ . Thus the sum of the amortized costs of all operations except *create-heap*, *insert* on a black node, and *meld* where both roots are black or at least one of the two heaps is entirely composed of black nodes is  $O(\sum_{i \in D} \log n_i)$ . There can not be more than  $|C|$  insertions of black nodes, *meld* operations where both roots are black or at least one of the two heaps is entirely composed of black nodes, or *create-heap* operations, all of which we have shown take constant amortized time. Therefore, the sum of the amortized times of all of the operations is at most  $O(|C| + \sum_{i \in D} \log n_i)$ . Adding the maximum potential drop of  $O(|C|)$  to the sum of the amortized costs yields a bound of  $O(|C| + \sum_{i \in D} \log n_i)$  on the actual runtime of the execution sequence  $S$ .

## 4 The Working Set Theorem

**Terminology:** Let  $t_i(x) = 1 +$  the number of items in the heap at time  $i$  that were not in the heap when  $x$  was inserted.<sup>2</sup> If time  $i$  is before  $x$  was inserted,  $t_i(x) = 0$ . Let  $T_i(x) = \max_{j=1}^i t_j(x)$ . Furthermore, assuming  $x$  was inserted at time  $a$  and the maximum size of the heap from time  $a$  to  $b$  is  $n$ ,  $T_b(x) \leq \min(b - a, n)$ . The value of  $T_i(x)$  is nondecreasing in  $i$ . We use  $n_i$  to denote the current number of nodes in the heap and  $\eta_i$  to denote the maximum value of  $T_j(x)$  among all nodes  $x$ , among all times  $j$  up to the and including  $i$ . Note that  $\eta_i$  is also equal to the maximum size the of the heap up to and including time  $i$ . In this section the standard tree terminology refers exclusively to the binary representation.

### Working Set Theorem for Pairing Heaps:

Let  $A = a_1 \dots a_m$  be a sequence of  $m$  *insert* and *extract-min* operations performed on an initially empty heap. Let  $I = \{i | a_i \text{ is a } \textit{insert} \text{ operation}\}$  and  $E = \{i | a_i \text{ is a } \textit{extract min} \text{ operation}\}$ . Let  $r_i$  denote the root at time  $i$ ; Thus if  $e \in E$ , then  $r_e$  is the item extracted at time  $e$ . The time to sequentially perform the operations in  $A$  on a pairing heap is:

$$O\left(n_m \log \eta_m + \sum_{i \in E} \log T_i(r_i)\right)$$

### Proof:

The potential method is used to analyze these operations. In order to analyze the pairing heap, we will introduce the notion of a dummy node. Each node will be inserted with a dummy node, and when extracted, the dummy node will either be extracted with it, or on the subsequent operation, if the subsequent operation is an *insert*. The dummy node of  $x$ , denoted as  $d(x)$  will be located as  $x$ 's rightmost child. The position of the dummy nodes is invariant under the pairing operation. The dummy nodes are for analysis purposes only, they are not stored in an implementation. In the node counting required to compute the  $t$  function defined above, we do not count dummy nodes. We instead adopt the convention that  $t_i(x) = t_i(d(x))$ .

**Lemma** For any fixed time  $i$ , there can never exist two nodes  $x$  and  $y$  such that  $T_i(x) = T_i(y)$ , unless one is the dummy node of the other.

**Proof** Assume  $x$  was inserted before  $y$ . Thus at any time  $j$  when both  $x$  and  $y$  are in the heap  $t_j(x) > t_j(y)$ . This is true because every node inserted after  $y$  that is still in the heap is a node that was inserted after  $x$  that was still in the heap, and  $y$  itself contributes 1 to  $t_j(x)$  but not to  $t_j(y)$ .

---

<sup>2</sup> Note that time  $i$  refers to the state of the data structure before the  $i$ th operation has completed. Time zero refers to the initial empty state of the data structure.

**Definitions**

$$\begin{aligned}
f(x) &= \frac{1}{2(1+x)^2} \\
S_i(x) &= \{y \mid y \text{ is in the subtree induced} \\
&\quad \text{by } x \text{ at time } i\} \\
r_i &= \text{the root} \\
\epsilon_i &= 2p_{i-1}(r_{i-1}) \\
&\quad \text{if } i-1 \in E \text{ and } i \in I, 0 \text{ otherwise} \\
r_i(x) &= \sum_{y \in S_i(x)} f(T_i(y)) \\
p_i(x) &= \log r_i(x) \\
\Phi_i &= \epsilon_i + \sum_{y \in S_i(r_i)} p_i(y) - 6n_i
\end{aligned}$$

The amortized time of operation  $i$ ,  $\hat{a}_i$ , is defined to be the actual time of the operation,  $a_i$ , plus the change in potential,  $\Phi_{i+1} - \Phi_i$ . Summing and rearranging terms yields  $\sum_{i=1}^{i=m} a_i = \sum_{i=1}^m \hat{a}_i + \Phi_0 - \Phi_{m+1}$ . Thus the actual runtime of a sequence of operations is equal to the sum of the amortized time of the operations plus the net loss of potential. To prove the working set theorem, it shall be sufficient to prove the following three lemmas:

**Lemma (Potential loss):**  $\Phi_1 - \Phi_{m+1} = O(n_m \log \eta_m)$

**Lemma (insert):** If  $i \in I$ ,  $\hat{a}_i = 0$ .

**Lemma (extract-min):** If  $i \in E$ ,  $\hat{a}_i = O(\log(T_i(r_i)))$

**Proof (Potential loss lemma):**

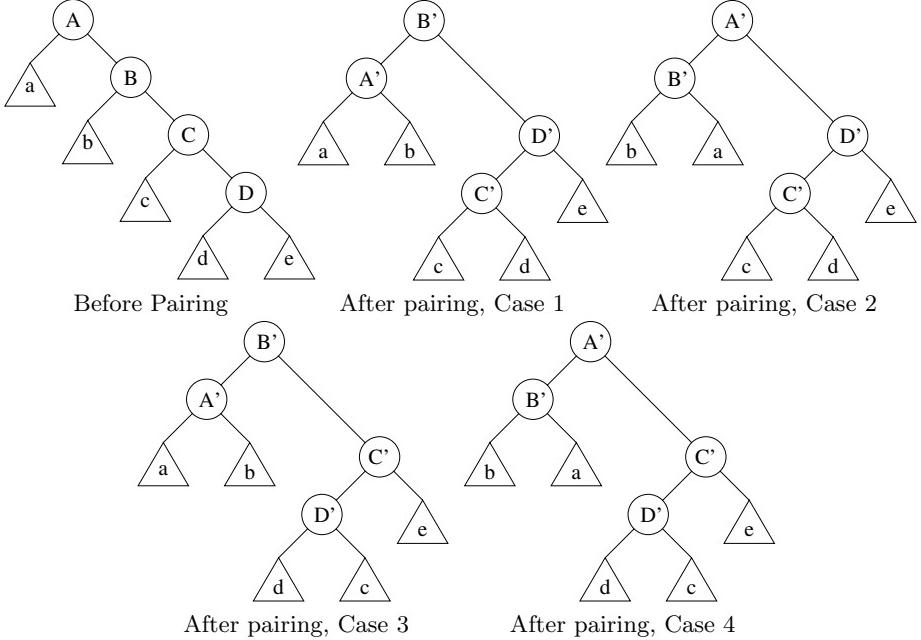
$$\begin{aligned}
&\Phi_1 - \Phi_{m+1} \\
&\leq \epsilon_1 - \epsilon_{m+1} + \sum_{x \in S_1(r_1)} p_1(x) \\
&\quad - \sum_{x \in S_{m+1}(r_{m+1})} p_{m+1}(x) \\
&\leq - \sum_{x \in S_{m+1}(r_{m+1})} \log r_{m+1}(x) \\
&\leq -n_{m+1}(-2 \log(1 + \eta_{m+1}) - 1) \\
&= O(n_m \log \eta_m)
\end{aligned}$$

**Proof (Extract-min Lemma):** We track the change in potential over every step involved in the *extract-min* operation: The removal of the root, the first pairing pass, the second pairing pass, the removal of the dummy node, and the changing of  $\epsilon$ ,  $T(x)$ , and  $n$ . We assume the current operation is  $a_\tau$ .

**Removal of root:** The removal of the root causes a change in potential of

$$-p_\tau(r_\tau) = -\log r_\tau(r_\tau) \leq 1 + 2 \log(1 + T_\tau(r_\tau))$$

**First Pairing Pass:** The effect on the binary representation of the first paring pass may be viewed as a sequence of applications of the double pairing transformation illustrated in Figure 1, along with at most one application of a single pairing transformation from Figure 2.



**Fig. 1.** Effect of first pairing pass on two pairs of nodes

**Lemma:** A single application of a double pairing transformation causes a potential gain of at most  $4p_\tau(A) - 4p_\tau(B)$ .

**Proof:** The proof for the first case is provided, the remaining three are substantially similar.

$$\begin{aligned}
 & p_\tau(A') + p_\tau(B') + p_\tau(C') + p_\tau(D') \\
 & - p_\tau(A) - p_\tau(B) - p_\tau(C) - p_\tau(D) \\
 = & \log r_\tau(A') + \log r_\tau(C') + \log r_\tau(D') \\
 & - \log r_\tau(B) - \log r_\tau(C) - \log r_\tau(D) \\
 = & \log r_\tau(A') + \log r_\tau(D) + \log 4 \\
 & + \log r_\tau(C') + \log r_\tau(D') \\
 & - \log r_\tau(B) - \log r_\tau(C) - 2 \log r_\tau(D) - 2 \\
 \leq & \log(4r_\tau(A')r_\tau(D)) + 2 \log r_\tau(A) \\
 & - 4 \log r_\tau(D) - 2 \\
 \leq & 2 \log(r_\tau(A') + r_\tau(D)) \\
 & + 2 \log r_\tau(A) - 4 \log r_\tau(D) - 2 \\
 \leq & 4 \log r_\tau(A) - 4 \log r_\tau(D) - 2 \\
 \leq & 4p_\tau(A) - 4p_\tau(D) - 2
 \end{aligned}$$

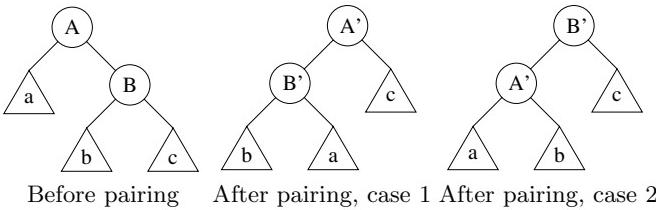
**Lemma:** A single application of a single pairing transformation from Figure 2 causes a potential gain of at most  $\log p_\tau(A) - \log p_\tau(B) \leq 4 \log p_\tau(A) - 4 \log p_\tau(B)$

**Proof:** Again, the proof for the first case is provided, with the second case being similar:

$$\begin{aligned} & p_\tau(A') - p_\tau(B') - p_\tau(A) - p_\tau(B) \\ &= p_\tau(B') - p_\tau(B) \\ &\leq p_\tau(A) - p_\tau(B) \\ &\leq 4p_\tau(A) - 4p_\tau(B) \end{aligned}$$

If the number of pairings required is even, then the first pairing pass is analyzed by repeatedly applying double transformations. If the number of pairings required is odd, the pairings are carried out by using one single-pairing transformation and several double-pairing transformations. Note that  $d(r_\tau)$ , which will be at the bottom of the right path, does not participate in the transformations, as it is not actually stored in the heap. Let  $l$  be the number of pairing transformations performed. All of the transformations applied form an telescoping sum with the result that the total potential gain is at most

$$\begin{aligned} & 4p_\tau(L(r_\tau)) - 4p_\tau(d(r_\tau)) - (l - 1) \\ &\leq 9 + 4 \log(T_\tau(d(r_\tau))) - l \\ &\leq 9 + 4 \log(T_\tau(r_\tau)) - l \end{aligned}$$



**Fig. 2.** Effect of a single pairing

**Second pairing pass:** The second pairing pass can be viewed as a sequence of single transformations. As stated above the potential change of each application of a single transformations transformation is  $\leq \log A - \log B$ . Repeatedly applying this transformation going up the right path generates a telescoping sum. Since the dummy node of the recently extracted node still lies at the bottom of

the right path, the sum is bounded by:

$$1 + 2 \log(T_\tau(r_\tau) + 1)$$

**Removal of dummy node:** Removing  $d(r_\tau)$  cases a potential gain of

$$\begin{aligned} -p_\tau(d(r_\tau)) &\leq 1 + 2 \log(1 + T_\tau(d(r_\tau))) \\ &= 1 + 2 \log(1 + T_\tau(r_\tau)) \end{aligned}$$

The dummy node is only removed if the next operation is not an insertion. In any event the change in potential caused by the possible removal of the dummy node is

$$\leq 1 + 2 \log(T_\tau(r_\tau) + 1)$$

**Setting of  $\epsilon_\tau$ :** Epsilon is assigned the value  $-2p_\tau(r_\tau)$  only if the next operation is an insert. As the previous value of  $\epsilon$  is zero, this causes a potential gain of at most

$$-2f(T_\tau(r_\tau)) \leq 2 + 4 \log(T(r_\tau) + 1))$$

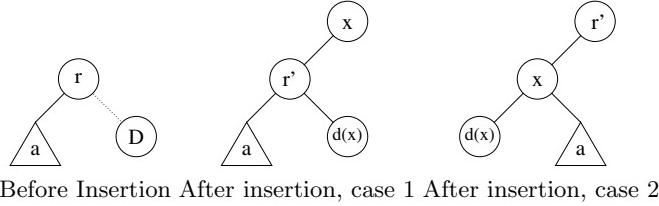
**Changing of  $T(x)$  and  $n$**  For all nodes  $x$  (except  $r_\tau$ , which has been removed),  $T_{\tau+1}(x) = T_\tau(x)$ . Thus no potential change due to the changing of the  $T$  values occurs. The removal of a nodes causes a potential gain of 6.

**Summary** The amortized cost of a remove min is the actual cost plus the change in potential. If we charge  $\frac{1}{2}$  unit of time for each pairing used, the actual cost is  $a \leq l + \frac{1}{2}$ . Note that by doubling the potential function it is possible to charge the more pleasing one unit of time per operation however, this was not done to simplify the presentation. Thus combining the actual cost with the changes in potential for the removal of the root, the first and second pairing passes, the removal of the dummy node and the possible setting of  $\epsilon$  yields:

$$\begin{aligned} \hat{a} &= a + \Phi_{\tau+1} - \Phi_\tau \\ &\leq l + \frac{1}{2} + 12 \log(T(r_\tau) + 1) + 12 - l \\ &\leq 12 \log(T_\tau(r_\tau) + 1) + \frac{25}{2} \\ &\leq O(\log T_\tau(r_\tau)) \end{aligned}$$

**Proof (Insert Lemma):** Inserting a new node  $x$  into a heap with root  $r_{\tau-1}$  will have two possible outcomes, depending on whether  $x$  is the smallest element of the new heap or not. This is depicted in Figure 3. Recall that if the previous operation was an *extract-max*, then the dummy of the previous root is still present, and  $\epsilon$  is nonzero. This possibility splits each of the two cases.

We may assume that the possible lowering of the values of  $r_\tau(x)$  caused by the possible increase of  $T_\tau(x)$  for some nodes  $x$ , has been performed. Note that the potential loss lemma limits the total potential decrease over a sequence of operations, and thus needs not be considered here. We also note the loss of 6 here caused by increasing  $n$  by one.



**Fig. 3.** Effect of insertion of a new element  $x$  in a pairing heap.  $D$  represents the dummy node that is present if the previous operation was an *extract-min*

**Previous operation was an *insert*:**

$$p_\tau(x) + p_\tau(r'_\tau) - p_\tau(r_\tau) \leq -p_\tau(r_\tau)$$

Since the last operation was an insert there is one element (either  $r$  or an element of  $A$ ) that has  $T(x) = 2$ . Thus

$$-p_\tau(r_\tau) \leq -\log f(T_\tau(2)) \leq \log 18$$

**Previous operation was an *extract-min*:** Removing the dummy causes a potential gain of  $-p_\tau(d(r_{\tau-1}))$ . Decreasing  $\epsilon_\tau$  to zero causes a potential gain of  $2p_{\tau-1}(d(r_{\tau-1}))$ . It is also the case that  $p_\tau(r) \geq p_\tau(d(r_{\tau-1}))$ . Thus the total potential gain is:

$$\begin{aligned} & p_\tau(r') + p_\tau(x) - p_\tau(r) - p_\tau(D) - \epsilon_\tau \\ &= -p_\tau(d(r_{\tau-1}) + -p_\tau(d(r_{\tau-1}) + +2p_{\tau-1}(d(r_{\tau-1})) \\ &\leq 0 \end{aligned}$$

**Summary:** Since the actual cost of the insert operation is  $a = 1$ , and the gain is potential is at most  $\log 18$  the amortized cost is:

$$\hat{a} = 1 + \log 18 - 6 \leq 0$$

#### 4.1 The Working Set Theorem for Top Down Skew Heaps

The working set theorem presented above can easily be adapted for seq-pairing heaps. Skew pairing heaps were introduced in [5], and implement all operations except *extract-min* identically to standard twopass pairing heaps. In the skew

pairing heap, *extract-min* pairs every other tree, incrementally from right to left. The remaining trees are also paired incrementally from right to left. Finally, the two resultant trees are paired together. In [5] it is shown that given any sequence of  $N$  operations (excluding (decrease-key) and (delete)) that takes time  $T$ , the same sequence can be executed on a top down skew heap in time  $T'$  where  $T \leq T' \leq n_N \log n_N$ . Since  $n_N \log n_N \leq \eta_N \log n_N$ , the same asymptotic bound of  $O(n_N \log \eta_N + \sum_{i \in E} \log T_i(r_i))$  holds for executing the sequence on a top down skew heap.

## 4.2 Populate Replace Heaps

Define a populate-replace-heap to be an abstract data structure that supports the following two operations:

*Populate*: Given  $n$  items, populate inserts them into the heap.

*Replace-Min*: Replace the minimum element with another.

**Theorem:** Define  $s(x)$  to be the number of items smaller than  $x$  when  $x$  is inserted into a heap. In populate replace heap, implemented as a pairing heap, threepass pairing, skew pairing or top down skew heap, one *populate* operation with  $n$  items  $x_1 \dots x_n$ , followed by  $N$  *replace-min* operations, where  $y_1 \dots y_N$  are the replacement items takes time  $O(n \log n + \sum_{i=1}^N \log s(y_i))$

**Proof:** Two observations: First,  $n_T \leq n$  throughout the life of the heap. Secondly, if when item  $x$  is inserted into a heap of  $n$  items there were  $k$  items smaller than  $x$  then there will never be more than  $n$  items inserted after  $x$  in the heap concurrently with  $x$ , if the heap size never exceeds  $n$ . This is because the  $n - k$  items larger than  $x$  upon insertion will still be present at  $x$ 's removal. Thus when  $x$  is removed  $T(x) \leq k$ . These two observations, along with the working set theorem for pairing heaps complete the proof. Note that constant time insertion in threepass pairing heaps yields a charge of  $n$  rather than  $n \log n$  in the above analysis.

## 4.3 {Pairing, Skew-Pairing, Top-Down Skew} Heaps Merge Sorted Lists Optimally within a Constant Factor

A populate replace heap, implemented as a pairing, skew-pairing, or top-down skew heap can be used to merge the items of  $m$  ordered lists of lengths  $n_1, n_2, \dots, n_m$  by storing one element of each list in a the heap, and then repeatedly removing the smallest item and replacing it with the next item in its list. Define  $k_{i,j}$  to be the number of items in the heap when  $j$ 'th item from list  $x_i$  is removed. Define  $N = \sum_{i=0}^m n_i$ . Note that  $\forall i \sum_{j=1}^{n_i} k_{i,j} \leq N$ .

**Theorem:** Pairing heaps merge  $m$  ordered lists size  $n_1, n_2 \dots n_m$  with a total of  $N$  elements optimally in time  $\Theta(m \log m + N + \sum_{i=1}^m n_i \log \left(\frac{N}{n_i}\right))$

**Lower bound**

**The Information theory bound**

Given a set of  $m$  ordered lists  $\{x_1, x_2 \dots x_n\}$ , where list  $i$  has  $n_i$  elements, we wish to generate one ordered list  $X$  with  $N = \sum_{i=1}^m n_i$  elements. Since

the number of possible orderings of  $X$  is given by the multinomial coefficient  $\binom{N}{n_1, n_2, \dots, n_m}$ , the information theoretic bound on this problem is

$$\log \binom{N}{n_1, n_2, \dots, n_m} \text{ which is } \Omega\left(-N \log e + \sum_{i=1}^m n_i \log\left(\frac{N}{n_i}\right)\right)$$

### Other lower bounds

We assume a lower bound of  $\Omega(N)$ , since each item must be looked it.

Since merging this lists sorts the  $m$  heads of each list there is an lower bound of  $\Omega(m \log m)$

### Summary

Linearly combining the three lower bounds above with suitable constants yields a lower bound of  $\Omega\left(m \log m + N + \sum_{i=1}^m n_i \log\left(\frac{N}{n_i}\right)\right)$

### Upper bound

The total cost of merging the lists according to the populate replace theorem is  $m \log m + \sum_{i=1}^m \sum_{j=1}^{n_i} \log k_{i,j} = O\left(m \log m + N + \sum_{i=1}^m n_i \log\left(\frac{N}{n_i}\right)\right)$

## References

1. R. Cole. On the dynamic finger conjecture for splay trees. part ii: The proof. Technical Report Computer Science TR1995-701, New York University, 1995.
2. R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the dynamic finger conjecture for splay trees. part i: Splay sorting log n-block sequences. Technical Report Computer Science TR1995-700, New York University, 1995.
3. M. L. Fredman. Manuscript in preparation.
4. M. L. Fredman. On the efficiency of pairing heaps and related data structures. *JACM*, 46(4):473–501, 1999.
5. M. L. Fredman. A priority queue transform. In *Workshop on Algorithm Engineering*, pages 243–257, 1999. LNCS 1668.
6. M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1:111–129, 1986.
7. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their used in improved network optimization algorithms. *JACM*, 34:596–615, 1987.
8. Haim Kaplan and Robert E. Tarjan. New heap data structures. Technical Report Computer Science TR-597-99, Princeton University, 1999.
9. D. D. Sleator and R. E. Tarjan. Self-adjusting binary trees. *JACM*, 32:652–686, 1985.
10. D. D. Sleator and R. E. Tarjan. Self-adjusting heaps. *SIAM Journal of Computing*, 15:52–69, 1986.
11. J. T. Stasko and J. S. Vitter. Pairing heaps: experiments and analysis. *CACM*, 15:234–249, 1987.
12. R. Sundar. *Amortized Complexity of Data Structures*. PhD thesis, New York University, 1991.
13. R. E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5:367–378, 1985.

# Maintaining Center and Median in Dynamic Trees

Stephen Alstrup<sup>1</sup>, Jacob Holm<sup>1</sup>, and Mikkel Thorup<sup>2</sup>

<sup>1</sup> The IT University of Copenhagen, Glentevej 67, DK-2400, Denmark.

{stephen, jholm}@itu.dk

<sup>2</sup> AT&T Labs–Research, mthorup@research.att.com

**Abstract.** We show how to maintain centers and medians for a collection of dynamic trees where edges may be inserted and deleted and node and edge weights may be changed. All updates are supported in  $O(\log n)$  time, where  $n$  is the size of the tree(s) involved in the update.

## 1 Introduction

In this paper we study the problem of locating facilities in a collection of dynamic trees. For each tree, we wish to maintain (1) a *center*, which is a node minimizing the maximal distance to all other nodes, and (2) a *median*, minimizing the sum of the distances to all other nodes. In both cases, we have edge weights, and in the later case, it is relevant to have node weights, and then the cost of the median is the weighted sum of distances to all other nodes.

In 1971 Goldman [17] gave a linear time algorithm for determining the *median* in a tree. In 1973 Handler [19] showed how one in linear time can compute the *center* of a tree. The static median and center problems have been investigated and generalized in many papers, see e.g. [18, 3, 15, 11]. A long list of references to the median and center problem and similar problems can be found in [26].

In our dynamic setting, we allow weights to be changed, and further edges may be inserted and deleted. In the rest of this paper,  $n$  denotes the size of the tree(s) involved in an operation. Our main result is that both centers and medians can be maintained in  $O(\log n)$  time per update. For centers, the previous bound was  $O(\log^2 n)$  [6]. For medians, polylogarithmic bounds were only known for changes of node weights [3], but not for edge insertions and deletions. More precisely, [3] presents an  $O(\log n)$  bound for the monotone case where weights may only be increased. If the weights may be both increased and decreased, they claim an  $O(\log^2 n)$  bound. However, to achieve these results, they claim they can access subtree weights in constant time, spending  $O(\log n)$  per weight update [3, p. 445]. This contradicts a cell-probe lower bound (for word size  $\log n$ ) saying that an update time of  $t_u$  implies a query time of  $\Omega(\log n / \log(t_u \log^2 n))$ , even if the tree is just a path (prefix-sum) [13, p. 348 (2)]. Our  $O(\log n)$  solution to the dynamic median problem does not maintain all subtree weights. All our algorithms are elementary in that they can be implemented on a pointer machine [28].

A common problem in finding medians and centers are that they are “non-local” properties. Here, by a *local property* we mean that if a node has the property in a tree, then it has the property in all subtrees it appears in. Local properties lend themselves nicely to bottom-up computations, whereas non-local properties tend to be more challenging.

A main advantage to our solutions of the dynamic center and median problems are that they are simple, based on the top trees from [2]. Towards the end of the paper, we will argue that it would have been more technical to solve the problem with the classical dynamic trees from [27]. Thus, our methodological contribution is to pin-point advantages of designing dynamic tree algorithms with top trees.

## 2 Top Trees

In this preliminary section, we discuss our basic starting point: top trees from [2]. Our presentation of the interface will be somewhat more precise and thorough than that in [2]. The more exact understanding of the interface is needed for both our applications, and for our later methodological discussion of top trees versus more classical data structures for dynamic trees [10, 12, 27].

A top tree is defined based on a pair consisting of a tree  $T$  and a set  $\partial T$  of at most 2 nodes from  $T$ , called *external boundary nodes*. Given  $(T, \partial T)$ , any connected subtree  $C$  of  $T$  has a set  $\partial_{(T, \partial T)} C$  of *boundary nodes* which are the nodes of  $C$  that are either in  $\partial T$  or incident to an edge in  $T$  leaving  $C$ . The subtree  $C$  is called a *cluster* of  $(T, \partial T)$  if it has at most two boundary nodes. Then  $T$  is itself a cluster with  $\partial_{(T, \partial T)} T = \partial T$ . Also, if  $A$  is a subtree of  $C$ ,  $\partial_{(C, \partial_{(T, \partial T)} C)} A = \partial_{(T, \partial T)} A$ , so  $A$  is a cluster of  $(C, \partial_{(T, \partial T)} C)$  if and only if  $A$  is a cluster of  $(T, \partial T)$ . Since  $\partial_{(T, \partial T)}$  is a canonical generalization of  $\partial$  from  $T$  to all subtrees of  $T$ , we will use  $\partial$  as a shorthand for  $\partial_{(T, \partial T)}$  in the rest of the paper.

A *top tree*  $\mathcal{T}$  over  $(T, \partial T)$  is a binary tree such that:

1. The nodes of  $\mathcal{T}$  are clusters of  $(T, \partial T)$ .
2. The leaves of  $\mathcal{T}$  are the edges of  $T$ .
3. If  $C$  is an internal node of  $\mathcal{T}$  with children  $A$  and  $B$ , then  $C = A \cup B$  and  $A$  and  $B$  are *neighbors*, that is they share a single node (see Figure II).
4. The root of  $\mathcal{T}$  is  $T$  itself.

A tree with a single node has an empty top tree.

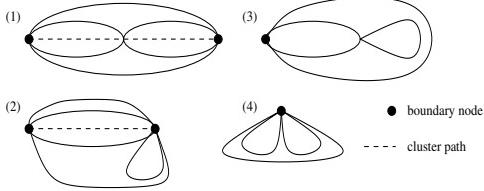
The top trees over the trees in our forest are maintained under the following operations:

**Link( $v, w$ ):** where  $v$  and  $w$  are in different trees, links these trees by adding the edge  $(v, w)$  to our dynamic forest.

**Cut( $e$ ):** removes the edge  $e$  from our dynamic forest.

**Expose( $v, w$ ):** where  $v$  and  $w$  are in the same tree  $T$ , makes  $v$  and  $w$  the external boundary nodes of  $T$ . Moreover, Expose returns the new root cluster of the top tree over  $T$ .

Expose can also be called with one or zero nodes as arguments if we want less than two external boundary nodes.



**Fig. 1.** Combining two clusters in one. The boundary nodes and cluster paths in the figure are for the resulting cluster.

In general, Link and Cut makes the set of external boundary nodes for the resulting trees empty. Every update of the top tree can be implemented as a sequence of the following two operations:

**Merge( $A, B$ ):** where  $A$  and  $B$  are the root-clusters of two top trees  $\mathcal{T}_A$  and  $\mathcal{T}_B$ .

Creates a new cluster  $C = A \cup B$  and makes it the common root of  $A$  and  $B$ , thus turning  $\mathcal{T}_A$  and  $\mathcal{T}_B$  into a single new top tree  $\mathcal{T}_C$ . Finally, the new root cluster  $C$  is returned.

**Split( $C$ ):** where  $C$  is the root-cluster of a top tree  $\mathcal{T}_C$  and has children  $A$  and  $B$ .

Deletes  $C$ , thus turning  $\mathcal{T}_C$  into the two top trees  $\mathcal{T}_A$  and  $\mathcal{T}_B$ .

Recall that  $n$  denotes the size of the trees involved in a given update operation. From [210] we have:

**Theorem 1** *For a dynamic forest we can maintain top trees of height  $O(\log n)$  supporting each Link, Cut, or Expose with a sequence of  $O(\log n)$  Merge and Split. Here the sequence itself is identified in  $O(\log n)$  time. The space usage of the top trees is linear in the size of the dynamic forest.*

Note that since the height is maintained logarithmic, any edge is contained in at most  $O(\log n)$  clusters. In contrast, a node  $v$  can appear in  $\Theta(n)$  clusters. However,  $v$  can only be a non-boundary node in  $O(\log n)$  clusters. More precisely, if  $v$  is not an external boundary node, there is a unique cluster  $C := \text{Merge}(A, B)$  where  $\{v\} = A \cap B$  and  $v \notin \partial C$ . Then  $v$  is a non-boundary node in a cluster  $D$  if and only if  $D = C$  or  $D$  is one of the  $O(\log n)$  ancestors to  $C$ .

*Notation* If  $v$  and  $w$  are connected in a tree,  $v \dots w$  denotes the unique path from  $v$  to  $w$ . If a cluster  $C$  has two boundary nodes  $a$  and  $b$ , we call  $a \dots b$  the *cluster path* of  $C$ , denoted  $\pi(C)$ . If  $|\partial(C)| < 2$ ,  $\pi(C) = \emptyset$ . Note that if  $A$  is a child cluster of  $C$  and  $A$  shares an edge with  $\pi(C)$ , then  $\pi(A) \subseteq \pi(C)$ , and then we call  $A$  a *path child* of  $C$ . In terms of boundary nodes, if  $C$  has children  $A$  and  $B$ ,  $A$  is a path child of  $A$  if and only if  $|\partial A| = 2$  and either  $\partial A = \partial C$  (Fig. II(2)) or  $\partial C \subset \partial A \cup \partial B$  (Fig. II(1)).

*Representation and usage of top trees* A top tree is represented as a standard binary rooted tree with parent and children pointers. The “top” nodes of the binary tree represent the clusters, and with each top node is associated the set of

at most two boundary nodes of the represented cluster. The leaves of the binary top tree are still identified with the edges of our tree. Finally, from each node  $v$ , there is a pointer  $C(v)$  to the smallest cluster that  $v$  is a non-boundary node in, or to the root cluster containing  $v$  if  $v$  is an external boundary node.

The user of the top tree data structure has direct access to the above representation, and will typically associate some extra information with the top nodes. The user is guaranteed that the top tree is only modified with Merge and Split. In connection with each Merge and Split the user is notified and given pointers to the top nodes representing the involved clusters. The user can then update his information associated with these top nodes.

For example, suppose, as in [27], that we want to maintain the minimum weight on the path between any two vertices. Then with each (top node representing a) cluster  $C$ , we store as extra information the minimum weight  $W_C$  on the cluster path  $\pi(C)$ . For an edge, this is just the edge weight. When  $C$  is created by a merge, we store the minimum weight stored at its path children. When  $C$  is split, we just discard the information stored with  $C$ . Now, to find the minimum weight between  $v$  and  $w$ , we set  $C := \text{Expose}(v, w)$ . Then  $\pi(C) = v \cdots w$ , and we return  $W_C$ .

Together with Theorem 1, the above description of how to modify and use our extra information  $W_C$  allows us to conclude that we can maintain a dynamic collection of trees with Cut, Link and queries to minimum weights between given nodes in  $O(\log n)$  time per operation.

We shall refer to the algorithm from Theorem 1 that translates Cut, Link, and Expose into sequences of Merge and Split as a *driver*. In our description of our extra information, we did not need to worry about how the driver scheduled the Merge and Split; we just had to tell how information had to be modified in connection with an arbitrary Merge and Split.

Above, Split was trivial. To see its relevance, suppose as in [27], that we as an additional operation want to add a weight  $x$  to all edges on a path  $v \cdots w$ . Then, for each cluster  $C$ , we introduce a “lazy” weight  $\Delta_C$  which is to be added to all edges in  $\pi(C)$  in all clusters strictly descending from  $C$ . The addition of  $x$  to  $v \cdots w$  is now done by calling  $C := \text{Expose}(v, w)$  and adding  $x$  to  $W_C$  and to  $\Delta_C$ . Then Split( $C$ ) requires that for each path child  $A$  of  $C$ , we set  $W_A := W_A + \Delta_C$  and  $\Delta_A := \Delta_A + \Delta_C$ . For  $C := \text{Merge}(A, B)$ , we set  $W_C := \min\{W_A, W_B\}$  and  $\Delta_C := 0$ . Finally, to find the minimum weight on the path  $v \cdots w$ , we set  $C := \text{Expose}(v, w)$  and return  $W_C$ .

Put in perspective, our top trees are natural generalizations of standard balanced binary trees over dynamic collections of lists that may be concatenated and split. In the balanced binary trees, each node represent a segment of a list, which in top terminology is just a special case of a cluster. Standard drivers for balanced binary trees also ascertain that the height is  $O(\log n)$ , and that each concatenation can be done by  $O(\log n)$  local modification, called rotations.

### 3 Non-local Searching

We are now going to build a black-box on top of our top trees for maintenance of centers and medians. As discussed in the introduction, the common feature of centers and medians is that they represent non-local properties. Here a node/edge property is local if it being satisfied by a node in a tree implies that the node satisfies the property in all subtrees containing it. For example, being the minimum edge on a given path is a local property. Local properties lend themselves nicely to bottom-up computations whereas non-local properties appear to be more challenging.

For our general non-local searching, the user should supply a function *Select* that given the root cluster of a topology tree, selects one of the two children. Recall here that a root cluster represents the whole underlying tree, which is important when dealing with non-local properties. Our black box will use *Select* to guide a binary search after a desired edge.

**Theorem 2 (Non-Local Search)** *Given a top tree, after  $O(\log n)$  calls to *Select*, *Merge*, and *Split*, there is a unique edge  $(v, w)$  contained in all clusters chosen by *Select*, and then  $(v, w)$  is returned.*

As stipulated in the general interface to top trees, the driver behind Theorem 2 will only manipulate the top tree with merge and split operations.

Before proving Theorem 2 we apply it to the center and median problems. Our general approach is to first decide the information needed for *Select*, second show how to make the information available.

#### 3.1 Dynamic Center

For any tree  $T$  and node  $v$  let  $h_v(T)$  denote the length of the longest path from  $v$  in  $T$ . A *center* is a node  $v$  minimizing  $h_v(T)$ .

**Lemma 3** *Let  $T$  be a tree, and let  $A$  and  $B$  be neighboring clusters with  $A \cap B = \{c\}$  and  $A \cup B = T$ . If  $h_c(A) \geq h_c(B)$ ,  $A$  contains all centers.*

**Proof:** Let  $w$  be a node in  $A$  of maximal distance to  $c$ . Then  $dist(c, w) = h_c(A) = h_c(T)$ . Now, if  $v \in B \setminus A$ ,  $h_v(T) \geq dist(v, w) = dist(v, c) + dist(c, w) = dist(v, c) + h_c(T)$ . Since the edge weights are positive,  $dist(v, c) > 0$ , so  $v$  cannot be a center minimizing  $h_v(T)$ .  $\square$

For every cluster  $C$ ,  $\partial C = \{a, b\}$ , we maintain:

- The distance between the boundary nodes:  $dist(C)$
- The maximal distance in  $C$  from each boundary node:  $h_a(C), h_b(C)$

Thus, for a new edge  $(v, w)$  with weight  $x$ , we just set

$dist((v, w)), h_v((v, w)), h_w((v, w)) := x$ . Consider merging  $A$  and  $B$  in  $C$ . To get  $dist(C)$  we just sum  $dist(D)$  for each path child  $D \in \{A, B\}$  of  $C$  (In Fig. 1, we have two path children in (1), one in (2), and none in (3,4)). For each  $a \in \partial C$ ,

we compute  $h_a(C)$  as a maximum over the two cluster children  $A$  and  $B$  of  $C$ , depending on which the node furthest from  $a$  is found in. If  $a \in \partial A$ , the maximal distance to a node in  $A$  is  $h_a(A)$ . If  $a \notin \partial A$  and  $\{c\} = A \cap B$ , we have to pass  $B$  to get to  $c$ , so the maximal distance to a node in  $A$  is  $dist(B) + h_c(A)$ . Splitting a node does not require moving any information, so we conclude that we can maintain the above information for the clusters in constant time per merge or split, hence in  $O(\log n)$  time per link or cut by Theorem 1.

We will now define Select given a root cluster  $C$  with children  $A$  and  $B$ ,  $A \cap B = \{c\}$ . If  $h_c(A) \geq h_c(B)$ , Select picks  $A$ , otherwise it picks  $B$ . By Lemma 3, any cluster picked contains all centers, so following Theorem 2, the returned edge  $(v, w)$  contains all centers. Since Select takes constant time,  $(v, w)$  is found in  $O(\log n)$  time.

To find out if whether  $v$  or  $w$  is a center, we compute  $C := \text{Expose}(v, w)$  in  $O(\log n)$  time, using Theorem 1. Since  $C$  coincides with  $T$ , we can return  $v$  if  $h_v(C) < h_w(C)$ ;  $w$  otherwise.

**Theorem 4** *The center can be maintained dynamically under link, cut and change of edge weights in  $O(\log n)$  worst case time per operation.*

**Proof:** Since the above Merge, Split, and Select are supported in constant time, the time bound follows from Theorem 2.  $\square$

### 3.2 Dynamic Median

Let  $T$  be a tree with both positive node and edge weights. A *median* is a node  $m$  minimizing  $\sum_{v \in V} weight(v) * dist(v, m)$ , where  $dist(v, m)$  is the sum of *cost* of edges on the unique path from  $v$  to  $m$  in the tree. For any tree  $T$ , let  $w(T)$  denote the sum of node weights of  $T$ . Our approach to finding medians is similar to that for centers, but for the median, it is natural to allow the user to change node weights, and this requires a simple trick.

The lemma below is implicit in Goldman [17].

**Lemma 5** *Let  $(v, w)$  be an edge in the weighted tree  $T$ . Let  $T_v$  and  $T_w$  be the trees from  $T \setminus \{(v, w)\}$  containing  $v$  and  $w$ , respectively. If  $w(T_v) = w(T_w)$ ,  $v$  and  $w$  are the only medians in  $T$ , and if  $w(T_v) > w(T_w)$ , all medians in  $T$  are in  $T_v$ .*

**Corollary 6** *Let  $T$  be a tree, let  $m$  be a median of  $T$  and let  $A$  and  $B$  be neighboring clusters with  $A \cap B = \{c\}$  and  $A \cup B = T$ . Then  $w(A) \geq w(B) \Rightarrow m \in A$ .*

**Proof:** Let  $(c, w)$  be any edge in  $B$  leaving  $c$ . Then  $T_c = A$  and  $w(T_c) = w(A) \geq w(B) > w(T_w)$ , so by Lemma 5 there are no medians in  $T_w$ . It follows that there cannot be any medians in  $B \setminus \{c\}$ .  $\square$

The above corollary suggest that we should maintain the weight of each cluster, but this gives rise to a problem; namely that a single node can be contained

in arbitrarily many clusters, and a change in the node's weight would affect all these clusters.

Our solution is that for each cluster  $C$ , we only maintain their “internal weight”  $w^i(C) = w(C \setminus \partial C)$ . This means that when we merge two clusters  $A$  and  $B$ ,  $A \cap B = \{c\}$  into  $C$ , we add their internal weights plus the weight of  $c$  if  $c \notin \partial C$ . The point is that any node is only non-boundary in  $O(\log n)$  clusters, and the internal weight of these clusters is trivially updated in  $O(\log n)$  time if a node weight changes.

Since clusters have at most two boundary nodes, for a given cluster, we can always compute the real weight from the internal weights in constant time, and hence we can implement Select choosing the lightest cluster in constant time, getting an edge  $(v, w)$  which contains the median in  $O(\log n)$  time.

To find out which of  $v$  and  $w$  is the median, we apply Lemma 5. We now cut the edge  $(v, w)$ , and return  $v$  if the (root cluster of the) tree  $T_v$  containing  $v$  is heavier;  $w$  otherwise. Before returning  $v$  and  $w$ , we link  $(v, w)$  back in  $T$ . The link and cut take  $O(\log n)$  time, so we conclude:

**Theorem 7** *The median can be maintained dynamically under link, cut and change of edge/node weights in  $O(\log n)$  worst case time per operation.*

### 3.3 Non-local Search Implementation

We will now prove Theorem 2. Essentially our search will follow a path down the given top tree  $\mathcal{T}$ . As we search down, we will modify the top tree so as to facilitate calls to select, but we will end up restoring it in its original form. Thus, when we start the search, we assume that some driver, as in Theorem 1, provides a top tree of height  $O(\log n)$ . It is convenient to assume that there is at least one external boundary node so that all clusters have at least one boundary node. During the search, we manipulate the top tree, but we end up returning it to the driver in exactly the same shape as we got it. All modifications for the search are done via Split and Merge, as stipulated in the general interface to top trees.

Our search consists of  $O(\log n)$  iterations  $i = 0, \dots$ . At the beginning of iteration  $i$ , there will be a cluster  $C_i$  on depth  $i$  in the original top tree which contains exactly the edges that have been in all clusters selected so far. If  $C_i$  is a single edge  $(v, w)$ , we return  $(v, w)$ . Otherwise  $C_i$  has children  $A_i$  and  $B_i$  in the original tree. Select will then be presented a root cluster with children  $A_i^*$  and  $B_i^*$  such that  $A_i \subseteq A_i^*$  and  $B_i \subseteq B_i^*$ . If the user selects  $A_i^*$ , we have  $C_{i+1} = A_i$  for the next iteration. Otherwise  $C_{i+1} = B_i$ .

To simplify the description of the generation of  $A_i^*$  and  $B_i^*$ , define the top tree of a singleton node  $c$  to be the cluster consisting of that node. Moreover, if  $c$  is a boundary node of a cluster  $C$ , define a merge with  $c$  to be neutral for  $C$ , that is,  $C$  remains a root cluster which is returned by the merge.

Now, let  $\partial C_i = \{a, b\}$ ,  $a \in A_i$ , and  $b \in B_i$ . Here, possibly,  $a = b$ . At the beginning of iteration  $i$ , we have three root clusters,  $C_i$ ,  $\hat{A}_i \ni a$ , and  $\hat{B}_i \ni b$ , partitioning  $T$  in the sense that they contain all edges and only overlap in  $a$  and

b. In the first iteration, we have  $C_0 = T$ ,  $\hat{A}_0 = \{a\}$  and  $\hat{B}_0 = \{b\}$ . For any  $i$ , we call the user-defined Select with the root cluster obtained as

$$\text{Merge}(\text{Merge}(\hat{A}_i, A_i), \text{Merge}(B_i, \hat{B}_i))$$

By symmetry, we may assume that the user selects  $\text{Merge}(\hat{A}_i, A_i)$ . We then split the newly created root cluster, as well as  $\text{Merge}(\hat{A}_i, A_i)$ , and then we have the three root clusters  $\hat{A}_{i+1} = \hat{A}_i$ ,  $C_{i+1} = A_i$ , and  $\hat{B}_{i+1} = \text{Merge}(B_i, \hat{B}_i)$  ready for iteration  $i + 1$ .

As mentioned, the iterations stop as soon as we arrive at a  $C_i$  which is just a single edge  $(v, w)$ . Since the height of the top tree before the search is  $O(\log n)$  and since each iteration only involves a constant number of Merge and Split, we conclude that the total number of Merge and Split is  $O(\log n)$ . At the end when we have found  $C_i = (v, w)$ , we just reverse all Merge and Split to restore the top tree in its original form, and return the edge  $(v, w)$ .

## 4 Methodological Remarks

Our main results in Theorem 4 and 7 could also have been achieved based on either the Sleator and Tarjan’s dynamic trees [27], or Frederickson’s topology trees [10,12]. However, we claim that the derivation from these more classical data structures would have been more technical.

*Sleator and Tarjan’s dynamic trees* Sleator and Tarjan provide an axiomatic interface for their dynamic trees [27] where the user can choose a root with a so-called Evert-operation, and then, for any specific node, add weights to all edges on the path to the root, or ask for the minimum of all weights on this path. This is basically the interface we implemented with top trees at the end of Section 2, assuming that we expose both the desired root and the specified node.

Before discussing limitations to the above interface, we first illustrate its generality viewing the min-query as representing an arbitrary associative operator  $\oplus$ . Suppose, for example, as in [27] that we want to implement parent pointers to the current root. We then let the weight of an edge be its pair of end-points and define  $a \oplus b = a$ . Then the “min”-query returns the end-points of the first edge on the path to the root, from which we immediately get a parent pointer.

Unfortunately, the above axiomatic interface has been found too limited for many application of dynamic trees, and instead authors have worked directly with the Sleator and Tarjan’s underlying representation [30,5,21,24,23,14,4,11,16,8,7,9,22]. In particular, this is the case for the previous solutions to the dynamic center [6] and median problems [3], and we believe part of the reason for their worse bounds and more complex solutions is difficulties in working directly with Sleator and Tarjan’s underlying representation.

Of course, one may try to increase the applicability of the axiomatic interface by augmenting it with further operations. For example, [25] shows how to find a

minimum weight node in a subtree. However, dealing with non-local properties is not so immediate, and we find it unlikely that we will ever converge to a set of operations so big that we can forget about the underlying representation.

The approach with top trees has instead been concentrated on designing a representation which is very easy to deal with directly. For example, to compute the minimum node of a given subtree as in [25], since we can insert and delete edges, this is equivalent to maintaining the minimum node of each tree in a dynamic forest, and this is again done by maintaining, for each cluster, the minimum weight over its non-boundary nodes. Since each node is only non-boundary in  $O(\log n)$  clusters, weight changes of nodes are trivially supported. If we do not expose any external boundary nodes, the root cluster will store the desired minimum.

*Frederickson’s topology trees* Top trees are very similar to Frederickson’s topology trees [10, 12], from which they were originally derived. The essential difference is that the clusters of topology trees are not connected via nodes, but via edges. Since Frederickson’s boundary consists of edges, he cannot have bounded boundaries for unbounded degree trees. Thus, in applications for unbounded degrees one has to code these with ternary trees; a quite standard process whose validity has to be verified for the individual application. Even if we assume we are dealing with ternary trees, topology trees still have clusters with up to three boundary edges instead of just two boundary nodes. Also topology merge combines two clusters *plus* the edge between them whereas a top merge just unites two neighboring clusters. Neither of these issues lead to fundamental difficulties, but in our experience, they lead to significantly more cases.

*Henzinger and King’s ET-trees* For completeness, we also mention Henzinger and King’s ET-trees [20]. This is a standard binary trees over the Euler tour of a tree. This technique is much simpler to implement than those mentioned above, and it can be used whenever we are interested in maintaining a min over the edges or nodes of a tree, where the min may be interpreted as any associative and commutative operation. Thus, the above mentioned result from [25] on maintaining the minimum weight node of a tree is immediate, and in fact, this was pointed out before [25] in [29]. However, the ET-trees cannot be used to maintain any of the path information discussed so far. Also, they cannot be used to maintain medians and centers.

## 5 Concluding Remarks

We have presented new and better bounds for maintaining medians and centers in dynamic trees. The results were obtained based on top trees, and we have argued more generally, that top trees in many instances would be the preferred data structure for solving problems on dynamic trees.

A top driver as described in Theorem 1 can be implemented by reduction to either Sleator and Tarjan’s techniques for dynamic trees [27], or Frederickson’s

techniques for topology trees [12]. The later option was pointed out in [2]. We are currently experimenting with these different implementations to see which leads to the better performance.

## References

1. A. Kanevsky, R. Tamassia, G. Battista, and J. Chen. On-line maintenance of the four-connected components of a graph (extended abstract). In *32nd FOCS*, pages 793–801, 1991.
2. S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *ICALP'97*, pages 270–280, 1997.
3. V. Auletta, D. Parente, and G. Persiano. Dynamic and static algorithms for optimal placement of resources in a tree. *TCS*, 165:441–461, 1996.
4. G. Battista and R. Tamassia. Incremental planarity testing (extended abstract). In *30th FOCS*, pages 436–441, 1989.
5. G. Battista and R. Tamassia. On-line graph algorithms with SPQR-trees. In *ICALP'90*, pages 598–611, 1990.
6. S. Cheng and M. Ng. Isomorphism testing and display of symmetries in dynamic trees. In *Proc. 7th SODA*, 1996.
7. R. Cohen and R. Tamassia. Dynamic expression trees. *Algorithmica*, 13(3):245–265, 1995.
8. R. Cohen and R. Tamassia. Combine and conquer. *Algorithmica*, 18(3):324–362, 1997.
9. R. F. Cohen and R. Tamassia. Dynamic expression trees and their applications. In *2nd SODA*, pages 52–61, 1991.
10. G. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SICOMP*, 14(4):781–798, 1985.
11. G. Frederickson. Parametric search and locating supply centers in trees. In *WADS'91*, volume 519 of *LNCS*, pages 299–319, 1991.
12. G. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and  $k$  smallest spanning trees. *SICOMP*, 26(2):484–538, 1997. See also FOCS'91.
13. M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st STOC*, pages 345–354, 1989.
14. Z. Galil and G. Italiano. Maintaining biconnected components of dynamic planar graphs. In *ICALP'91*, 1991.
15. B. Gavish and S. Sridhar. Computing the 2-median on tree networks in  $O(n \log n)$  time. *Networks*, 26, 1995. See also Networks Vol. 27, 1996.
16. A. Goldberg, M. Grigoriadis, and R. Tarjan. Use of dynamic trees in a network simplex algorithm for the maximum flow problem. *Math. Programming*, 50:277–290, 1991.
17. A. Goldman. Optimal center location in simple networks. *Transportation Sci.*, 5:212–221, 1971.
18. S. Hakimi and O. Kariv. An algorithmic approach to network location problems. ii: the p-medians. *SIAM J. APPL. MATH.*, 37(3):539–560, 1979.
19. G. Handler. Minimax location of a facility in an undirected tree network. *Transportation Sci.*, 7:287–293, 1973.
20. M. R. Henzinger and V. King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proc. 27th Symp. on Theory of Computing*, pages 519–527, 1995.

21. J. A. La Poutré. *Dynamic graph algorithms and data structures*. PhD thesis, Dep. Comp. Sci., Utrecht Uni., 1991.
22. S. Peckham. Maintaining tree projections in amortized  $O(\log n)$  time. Technical Report TR89-1034, Cornell Uni., Comp. Sci. Dep., 1989.
23. J. A. L. Poutré. Alpha-algorithms for incremental planarity testing. In *26'th STOC*, pages 706–715, 1994.
24. J. L. Poutré. Maintenance of triconnected components of graphs. In *ICALP'92*, volume 623 of *LNCS*, pages 354–365, 1992.
25. T. Radzik. Implementations of dynamic trees with in-subtree operations. *ACM J. Experimental Algorithmics*, 3:Article 9, 1998.
26. A. Rosenthal and J. Pino. A generalized algorithm for centrality problems on trees. *J. ACM*, 36:349–361, 1989.
27. D. Sleator and R. Tarjan. A data structure for dynamic trees. *JCSS*, 26(3):362–391, 1983. See also STOC'81.
28. R. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *Journal of computer and system sciences*, 18(2):110–127, 1979. See also STOC 1977.
29. R. Tarjan. Dynamic trees and search trees via euler tours, applied to the network simplex algorithm. Technical Report 503-95, Dep. Comp. Sci., Princeton Uni., September 1995.
30. J. Westbrook and R. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.

# Dynamic Planar Convex Hull with Optimal Query Time and $O(\log n \cdot \log \log n)$ Update Time

Gerth Stølting Brodal\* and Riko Jacob\*

BRICS\*\*, Department of Computer Science, University of Aarhus  
[{gerth,rjacob}@brics.dk](mailto:{gerth,rjacob}@brics.dk)

**Abstract.** The dynamic maintenance of the convex hull of a set of points in the plane is one of the most important problems in computational geometry. We present a data structure supporting point insertions in amortized  $O(\log n \cdot \log \log \log n)$  time, point deletions in amortized  $O(\log n \cdot \log \log n)$  time, and various queries about the convex hull in optimal  $O(\log n)$  worst-case time. The data structure requires  $O(n)$  space. Applications of the new dynamic convex hull data structure are improved deterministic algorithms for the  $k$ -level problem and the red–blue segment intersection problem where all red and all blue segments are connected.

## 1 Introduction

The problem of maintaining the convex hull of a set of points in the plane under the insertion and deletion of points is one of the foremost important problems in computational geometry [6,10]. A dynamic data structure for maintaining the convex hull of a point set has numerous applications, e.g. in algorithms solving the  $k$ -level problem [7] and the red–blue segment intersection problem where all red and all blue segments are connected [1]. For further applications see [4].

Overmars and van Leeuwen in 1981 gave a solution for the fully dynamic convex hull problem supporting point insertions and deletions in  $O(\log^2 n)$  time, where  $n$  is the maximum number of points in the set [12]. The data structure of Overmars and van Leeuwen stores the convex hull in a search tree and typical queries on the convex hull are supported in  $O(\log n)$  time. Preparata and Vitter gave a simpler approach achieving the same bounds as Overmars and van Leeuwen in [14]. Until recently there was made no progress on improving the update bounds for the general case. First in 1999, Chan presented a data structure that achieves amortized  $O(\log^{1+\varepsilon} n)$  update time, where  $\varepsilon > 0$  is any arbitrary constant, and  $O(\log n)$  query time for various types of queries, e.g. membership and tangent-finding [4].

For special cases better update bounds are known. For the semi-dynamic case where only insertions are allowed, it is easy to achieve  $O(\log n)$  insertion

\* Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

\*\* Basic Research in Computer Science, Centre of the Danish National Research Foundation.

time [13]. For the other semi-dynamic case where only deletions are allowed after preprocessing, Hershberger and Suri achieved  $O(n \log n)$  preprocessing time and amortized  $O(\log n)$  deletion time [9]. For the off-line case where the sequence of updates is given in advance, a data structure using  $O(n \log n)$  time for processing a sequence of  $n$  updates was given in [10]. The case where the sequence of updates is random was considered in [11][15], where it was shown how to achieve expected  $O(\log n)$  update time.

In this paper, we first give a new data structure for the semi-dynamic problem where only deletions are allowed after preprocessing, by extending the construction of Hershberger and Suri [9]. Provided that the initial point set is given lexicographically sorted, we achieve amortized  $O(n)$  preprocessing time, and amortized  $O(\log n \cdot \log \log n)$  deletion time. The data structure requires  $O(n)$  space. Our main result for the fully dynamic case is a transformation strategy that combines a fully dynamic data structure with a semi-dynamic data structure for the deletions only case, and generates a new fully dynamic data structure. The construction is based on the construction of Chan [4] combined with several new ideas. Let  $U(n)$  and  $D(n)$  be two nondecreasing positive functions, where  $U(n) \geq \log n$  and  $D(n) \geq \log n$ . If there exists a fully dynamic data structure with amortized  $O(U(n))$  update time and worst-case  $O(\log n)$  query time, and a semi-dynamic data structure with  $O(n)$  preprocessing time and amortized  $O(D(n))$  deletion time, then the transformation yields a data structure with amortized  $O(U(\log^4 n) \cdot \log n / \log \log n)$  insertion time, amortized  $O(D(n))$  deletion time, and worst-case  $O(\log n)$  query time. The queries that can be supported are: find the extreme point on the convex hull in a given direction; report whether a given line intersects the convex hull; report if a given point is contained in the interior of the convex hull; find the two points adjacent to a point on the convex hull; and given an exterior point find the two tangent points on the convex hull from the point.

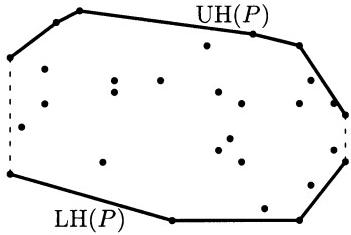
Combining our semi-dynamic data structure with the fully dynamic data structure of Overmars and van Leeuwen [12], we immediately get amortized  $O(\log n \cdot \log \log n)$  deletion and insertion time. By bootstrapping, we can use the resulting data structure as the fully dynamic data structure in the construction and the insertion time reduces to amortized  $O(\log n \cdot \log \log \log n)$  time, while the deletion time remains amortized  $O(\log n \cdot \log \log n)$ .

We note that a semi-dynamic data structure with  $O(n)$  preprocessing time and  $O(\log n)$  deletion time, would for any constant  $k$  imply a fully dynamic data structure with amortized  $O(\log n \cdot \log^{(k)} n)$  insertion time and amortized  $O(\log n)$  deletion and worst-case  $O(\log n)$  query time, by  $k - 1$  applications of our transformation strategy and using the data structure of Overmars and van Leeuwen as the initial fully dynamic data structure.<sup>1</sup>

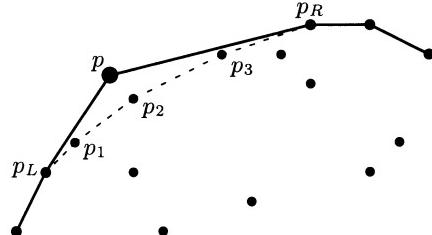
The paper is organized as follows. Section 2 contains a description of the semi-dynamic data structure for the deletions only case, and Sect. 3 and 4 contain the results for the fully dynamic case. Section 5 gives applications of the fully dynamic data structure.

---

<sup>1</sup> We let  $\log^{(1)} n = \log n$ , and  $\log^{(i+1)} n = \log \log^{(i)} n$  for  $i \geq 1$ .



**Fig. 1.** The convex hull  $\text{CH}(P)$  of a set of points  $P$  can be partitioned into an upper hull  $\text{UH}(P)$ , a lower hull  $\text{LH}(P)$ , and possibly two vertical lines.



**Fig. 2.** Deletion of the point  $p$  from the upper hull implies that  $p$  is replaced by the sequence of points  $p_1, p_2, p_3$ .

## Notation

Given a set of points  $P$  in the Euclidean plane, we let  $\text{CH}(P) \subseteq P$  denote the set of points on the convex hull of  $P$ , and  $\text{UH}(P)$  and  $\text{LH}(P)$  denote respectively the upper and lower hull of  $\text{CH}(P)$ . Figure 1 shows the upper and lower hulls of a set of points. In the following we restrict our attention to the upper hulls of the sets of points, and assume for the sake of simplicity that points are in general position, i.e. all points have distinct  $x$ -coordinates and no three points are on a line. The results for the convex hull problems immediately follow from the results on the upper hulls.

## 2 Semi-dynamic Data Structure

In this section we give a data structure for the semi-dynamic problem with amortized  $O(n)$  preprocessing time, and which supports point deletions in amortized  $O(\log n \cdot \log \log n)$  time. To achieve linear preprocessing time we require points to be given lexicographically sorted. The data structure supports the operations:

**Build** Given a lexicographically sorted set  $P$  containing  $n$  points, builds a data structure for  $P$  and returns the points on  $\text{UH}(P)$  from left-to-right.

**Delete** Deletes a point  $p$  from  $P$ , and returns the changes to  $\text{UH}(P)$ , i.e. if  $p$  was contained in  $\text{UH}(P)$  before the deletion then the sequence of new points on  $\text{UH}(P)$  are returned from left-to-right (see Fig. 2).

Our result for the semi-dynamic problem is the following.

**Theorem 1.** *There exists a data structure supporting Build in amortized  $O(n)$  time and Delete in amortized  $O(\log n \cdot \log \log n)$  time. The data structure requires  $O(n)$  space.*

In the following we without loss of generality assume  $n \geq 4$ , such that  $\log \log n \geq 1$ . Let  $P = \{p_1, p_2, \dots, p_n\}$  be the initial set of points, where  $p_i < p_{i+1}$  for  $1 \leq i < n$ , and let  $B = \lceil \log n \rceil$  and  $N = \lceil n/B \rceil$ . We partition  $P$  into a sequence of blocks  $P_1, \dots, P_N$ , each of size  $B$  except for  $P_N$ , where  $P_i =$

$\{p_{1+(i-1)B}, p_{2+(i-1)B}, \dots, p_{\min(iB, n)}\}$ , for  $1 \leq i \leq N$ . After a sequence of Delete operations we let  $\bar{P} \subseteq P$  denote the set of points which have not been deleted so far, and similarly we for  $P_1, \dots, P_N$  define  $\bar{P}_1, \dots, \bar{P}_N$ .

For each block  $P_i$ , the points  $\bar{P}_i$  are stored in sorted order in a linked list,  $\text{UH}(\bar{P}_i)$  is stored as a perfect balanced binary tree, and furthermore the points from left-to-right on  $\text{UH}(\bar{P}_i)$  are kept in a doubly linked list.

Since  $|\bar{P}_i| \leq B$ , the upper hull  $\text{UH}(\bar{P}_i)$  can be constructed by a linear sweep of  $\text{UH}(\bar{P}_i)$  in  $O(B)$  time, see e.g. [2, Sect. 1.1]. The balanced tree and the double linked list storing  $\text{UH}(\bar{P}_i)$  can therefore be recomputed in  $O(B)$  time, when a point is deleted from block  $P_i$ .

The blocks  $P_1, \dots, P_N$  are stored from left-to-right at the leaves of a perfect balanced binary tree  $T$  with height  $\lceil \log N \rceil$ . For each node  $v$  in  $T$ , we let  $T_v$  denote the subtree of  $T$  rooted at  $v$ , and let  $\bar{P}_v$  denote the union of the sets  $\bar{P}_i$  stored at the leaves of  $T_v$ . It is easy to see that  $\text{UH}(\bar{P}_v) \cap \text{UH}(\bar{P}_i)$  is either empty or a consecutive subsequence of  $\text{UH}(\bar{P}_i)$ . At each node  $v$  of  $T$  we store  $\text{UH}(\bar{P}_v)$  as a doubly linked list  $L_v$  of *block-records*, such that for each block  $P_i$  contributing to  $\text{UH}(\bar{P}_v)$ , i.e.  $\text{UH}(\bar{P}_v) \cap \text{UH}(\bar{P}_i) \neq \emptyset$ , we have a block-record  $r_{v,i}$ . For each block-record  $r_{v,i}$  we store pointers to the leftmost and rightmost points in  $\text{UH}(\bar{P}_i)$  which are also in  $\text{UH}(\bar{P}_v)$ . For a block  $P_i$ , let  $v_0, v_1, \dots, v_k$  be the prefix of the nodes in  $T$  on the path from the leaf  $v_0$  storing  $\bar{P}_i$  to the root, where  $\text{UH}(\bar{P}_i) \cap \text{UH}(\bar{P}_{v_j}) \neq \emptyset$ , i.e.  $r_{v_j,i} \in L_{v_j}$ . For  $0 \leq j < k$ , we with  $r_{v_j,i}$  store an *up-pointer* to  $r_{v_{j+1},i}$ . This representation allows us to efficiently navigate  $\text{UH}(\bar{P}_v)$  in both directions from point-to-point and block-to-block in constant time. Note that  $\text{UH}(\bar{P})$  is stored at the root of  $T$ .

Since each block requires  $O(B)$  space the total space for the  $N$  blocks is  $O(N \cdot B)$ . Since  $P$  is partitioned into  $N$  blocks, the total space for the lists of block-records at each level of  $T$  is at most  $O(N)$ . The total space required is  $O(N \cdot B + N \cdot \log N) = O(n)$ .

We now turn to the implementation of the operations. For Build the input set  $P$  is first partitioned into  $N$  blocks, and for each block the upper hull is computed by a sweep line algorithm in  $O(B)$  time and each block structure is initialized in  $O(B)$  time. The construction time for all blocks is  $O(n + N \cdot B) = O(n)$ . The tree  $T$  is then processed bottom-up level by level. Assume a node  $v$  has two children  $w_1$  and  $w_2$ , and  $L_{w_1}$  and  $L_{w_2}$  have already been computed (for a leaf  $\ell$ , we define  $L_\ell$  to only contain one block-record with pointers to the first and last node of  $\text{UH}(\bar{P}_\ell)$ ). First we let  $L_v$  be the concatenation of  $L_{w_1}$  and  $L_{w_2}$ . The resulting list of block-records represents a sequence of points forming a convex curve except for possible at one point, namely the last point from  $\text{CH}(\bar{P}_{w_1})$  or the first point from  $\text{CH}(\bar{P}_{w_2})$ , i.e. there is a pointer to  $p$  in one of the block records in  $L_v$ .

To fix this problem we apply the standard method used in convex hull construction algorithms: while we have a non-convex point  $p$  in the list of points, i.e.  $p$  together with its predecessor and successor point in the list form a left-turn, we remove  $p$  from the list. Removing  $p$  is done as follows: if  $p$  is in block  $P_i$ , and  $p$  is the only point from  $\text{UH}(\bar{P}_i)$  in the list, i.e. both pointers in  $r_{v,i}$  point

to  $p$ , we remove  $r_{v,i}$  from  $L_v$ . Otherwise we replace the pointer to  $p$  in  $r_{v,i}$  by a pointer to the next point in  $\text{UH}(\bar{P}_i)$  in the direction of the point given by the other pointer in  $r_{v,i}$ , where we utilize that the points in  $\text{UH}(\bar{P}_i)$  are kept in a double linked list. We can at most remove a point once in the bottom-up preprocessing of  $T$ , and the time for preprocessing one level of  $T$  is  $O(n)$  plus the time used to eliminate left turns. The total time for constructing all  $L_v$  lists becomes  $O(n + N \cdot \log N) = O(n)$ . It follows that Build takes  $O(n)$  time.

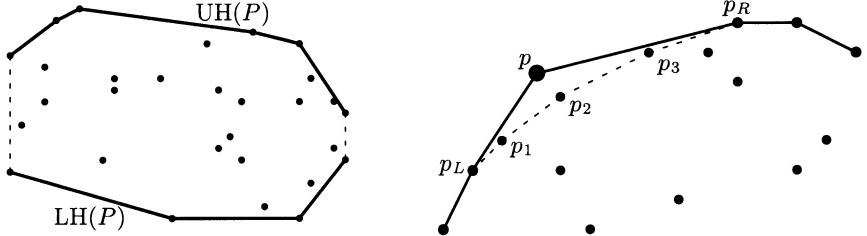
Before describing the Delete operation, we observe that only upper hulls actually containing  $p$  need to be updated (see Fig. 2). To perform Delete first in  $O(\log n)$  time make a binary search locating the block  $P_i$  containing  $p$ , assuming that  $P$  was given as an array of points or that we keep  $P$  in a balanced search tree. In  $O(B)$  time we check if  $p \in \text{UH}(\bar{P}_i)$ . If  $p \notin \text{UH}(\bar{P}_i)$  then no upper hull needs to be updated and it is sufficient to remove  $p$  from the list of points in  $\bar{P}_i$  in  $O(B)$  time. Otherwise  $p \in \text{UH}(\bar{P}_i)$ , and let  $\bar{p}$  and  $\bar{p}'$  be the predecessor and successor of  $p$  in  $\text{UH}(\bar{P}_i)$  (if present), and rebuild in  $O(B)$  time the data structure for block  $P_i$  after  $p$  has been deleted from the list of points in  $\bar{P}_i$ . What remains is to update all the upper hulls which contained  $p$ . If  $p \in \text{UH}(\bar{P}_v)$  for a node  $v$  then  $r_{v,i} \in L_v$ . But then  $r_{v,i}$  is reachable from  $\bar{P}_i$  using the stored up-pointers.

The reconstruction of upper hulls is done bottom-up in  $T$ . Consider a node  $v$  and the effect of deleting  $p$  from  $\text{UH}(\bar{P}_v)$ . Let  $p_L$  and  $p_R$  be the two points in  $\bar{P}_i$  that  $r_{v,i}$  has pointers to, where  $p_L \leq p_R$ . If  $p < p_L$  or  $p > p_R$  then  $p \notin \text{UH}(\bar{P}_v)$  and we are done. If  $p_L < p < p_R$  then the changes to  $\text{UH}(\bar{P}_v)$  can only be between  $p_L$  and  $p_R$ , i.e. the updates are done locally in block  $P_i$  and no changes are required for  $L_v$ . The complicated case is when  $p = p_L$  or  $p = p_R$ . First we need to delete  $p$  from the upper hull stored at  $v$ . If  $p_L = p_R$  then  $p$  was the only point from block  $P_i$ , and we delete  $r_{v,i}$  from  $L_v$ . Otherwise we have two cases: if  $p = p_L$  then we replace the pointer to  $p$  in  $r_{v,i}$  by a pointer to  $\bar{p}'$ , and if  $p = p_R$  then we replace the pointer to  $p$  in  $r_{v,i}$  by a pointer to  $\bar{p}$ .

After having deleted  $p$  from  $\text{UH}(\bar{P}_v)$ , we must insert new points onto  $\text{UH}(\bar{P}_v)$ , as illustrated by Fig. 2. If  $p$  was not an endpoint of the bridge connecting two points on the two upper hulls stored at the children of  $v$  (see Fig. 3), then the changes to  $\text{UH}(\bar{P}_v)$  are exactly the changes to  $\text{UH}(\bar{P}_w)$ , where  $w$  is the child of  $v$  where  $p \in \text{UH}(\bar{P}_w)$  before the deletion. It follows that it is sufficient to create and update existing block-records in  $L_v$  with exactly the same pointers to points in blocks as done for  $L_w$ .

The final case is when  $p$  is an endpoint of the bridge connecting the upper hulls stored at the children of  $v$ , ad illustrated in Fig. 3. Assuming the new bridge has been found, then updating  $L_v$  with respect to the new points on  $\text{UH}(\bar{P}_v)$  consists of inserting a subsequence of the points from each of the upper hulls stored at the children of  $v$ , by creating a sequence of new block-records in  $L_v$  with the same information as stored at the two children of  $v$  and changing at most four pointers in the block-records in  $L_v$  corresponding to the ends of the subsequences copied.

To find the new bridge we apply a standard bridge searching algorithm, with minor modifications. The standard bridge searching procedure keeps for



**Fig. 3.** The bridge between two horizontally separated upper hulls. The dashed lines show the changes to the left upper hull and the new bridge when deleting point  $p$ .

the upper hulls two candidate intervals for each of endpoints of the bridge, and performs a “simulations binary search” on both hulls, always halving at least one of the intervals. See e.g. [13, Lemma 3.1] for further details. We replace the binary search by a linear block search on each of the two upper hulls. The linear block search at the left child proceeds left-to-right, always trying to advance one block, whereas the linear block search at the right child proceeds right-to-left. Whenever a search is advanced to the next block a block-record is added to  $L_v$  in  $O(1)$  time.

The search process for each upper hull first tries to advance a complete block at a time, using the information stored at the block-records at the children of  $v$  to always pick the last point in the next block  $P_i$  contributing to  $\text{UH}(\bar{P}_i)$ . After having localized the block  $P_i$  containing one endpoint of the new bridge the search then proceeds in a binary fashion using the search tree storing  $\text{UH}(\bar{P}_i)$ . The total time for finding a bridge becomes linear in the number of block-records created plus  $O(\log B)$ . The output of `Delete` can be generated immediately from the changes to  $L_{\text{root}(T)}$ .

The total time for a deletion becomes  $O(B + x + \log N \cdot \log B)$ , where  $x$  is the total number of new block-records created. Since a deletion at most removes one block-record from each level of  $T$ , it follows that  $D$  deletions at most delete  $D \cdot \log N$  block-records. Since there can at most be  $O(N \cdot \log N)$  block-records, it follows that the total time for  $D$  deletions is at most  $O(D \cdot B + N \cdot \log N + D \cdot \log N + D \cdot \log N \cdot \log B) = O(n + D \cdot \log n \cdot \log \log n)$ . Since the  $O(n)$  term can be charged to `Build`, it follows that `Build` takes amortized  $O(n)$  time and each `Delete` operation amortized  $O(\log n \log \log n)$  time.

### 3 Fully Dynamic Data Structure

For this part of the paper we change the point of view of the exposition to the dual problem and consider upper envelopes instead of upper hulls. This duality, as explained e.g. in [2, p. 167], maps points to lines and vice versa in a way, that preserves above/on/below relations. In this setting a set of points becomes a collection of lines  $L$ , and the upper hull transforms to the upper envelope of these lines, i.e. the collection of line segments such that points on a segment

are not below any other line. An extreme point query, i.e. given a slope  $q$  find the point of the upper hull that has a tangent of slope  $q$ , turns into a vertical line query, i.e. given a vertical line with  $x$ -coordinate  $q$ , report the segment of the upper envelope crossing this line. Note that this is really only a change in point of view. There is no need to perform a computation to go from the original setting to the dual and back.

We apply a standard dynamization technique that divides the current points into sets and keeps one deletion only data structure per set. Additionally there is a more explicit representation of the current upper envelope, namely an interval tree, that allows fast queries without requiring too much work for updates. Inside the interval tree have at each internal node a fully dynamic upper envelope data structure, a so called *secondary structure*. The running time improvement relies on a polylogarithmic bound on the size of the secondary structures.

The description so far fits as well to the data structure proposed in Chan [4]. Compared to that data structure we apply improved deletion only data structures. We also do some explicit grouping of the subenvelopes stemming from the dynamization, such that the number of secondary structure storing segments from one subenvelope is reduced.

The remaining of this section is devoted to proving the following theorem.

**Theorem 2.** *Let  $U(n)$  and  $D(n)$  be two nondecreasing positive functions, where  $U(n) \geq \log n$  and  $D(n) \geq \log n$ . Assume there exists a data structure for the dynamic upper envelope problem supporting Insert and Delete in amortized  $O(U(s))$  time, and Vertical Line Query in worst-case  $O(\log s)$  time, where  $s$  is the total number of lines inserted. Assume further that there exists a data structure for semi-dynamic upper envelope problem supporting Build on a lexicographically sorted list of  $n$  points in amortized  $O(n)$  time and Delete in amortized  $O(D(n))$  time, where  $n$  is the number of lines in the structure.*

*Then there exists a data structure for the dynamic upper envelope problem supporting Insert in amortized  $O(\log n \cdot U(\log^4 n) / \log \log n)$  time and Delete in amortized  $O(D(n) + \log n \cdot U(\log^4 n) / \log \log n)$  time, and Vertical Line Query in worst-case  $O(\log n)$  time, where  $n$  is the total number of lines inserted.*

Applying this theorem to the data structure of Overmars and van Leeuwen with  $U(s) = \log^2 s$  and the result from Sect. 2 with  $D(n) = \log n \cdot \log \log n$ , we get Insert in  $O(\log n \cdot \log^2(\log^4 n) / \log \log n) = O(\log n \cdot \log \log n)$ , and Delete in  $O(\log n \cdot \log \log n)$ . Applying the theorem again on this new data structure improves Insert to  $O(\log n \cdot \log \log \log n)$ . The performance of the deletion only data structure is the bottleneck, that renders further applications of the theorem useless.

For the purpose of describing our data structure, we separate it into several layers. We first describe the layers in a top down fashion, we start with a data structure that solves the fully dynamic upper envelope problem using some auxiliary data structures. For the analysis we proceed in a bottom up fashion, i.e. we always analyze the auxiliary data structure first. This avoids any forward references.

### 3.1 The Interfaces

#### Fully dynamic upper envelopes.

**Insert** Insert a line, given by the parameters  $a$  and  $b$  in the representation  $y = ax + b$ . Return a pointer to a new line data structure.

**Delete** Given a pointer to a line data structure, delete that structure and the line it represents.

**Query** Given a value  $v$ , report the highest intersection of a line with the vertical line given by  $x = v$ .

**Query structure  $Q$ .** This data structure combines several independent upper envelopes. It is asserted (and could be easily checked), that the list of line segments in fact form envelopes. It is also asserted, that a line is present in at most one set and has therefore at most one segment.

There is an active set of segments that is considered for queries. For all lists of segments it is asserted, that the segments from this list form an upper envelope. A segment is given by a line and an interval on the  $x$ -axis.

**Init set with active envelope** Given a lexicographically sorted list  $L$  of lines and a list  $K \subseteq L$  of segments. Initialize a set data structure that can hold upper envelopes stemming from lines in  $L$  and insert  $K$  into the active set. It is asserted that  $K$  forms a complete upper envelope. Return a pointer to a new data structure representing the set.

**Delete set** Delete a set given by a pointer, removing all segments from the active set.

**Replace inside an envelope** Given a pointer to a set, pointers to (up to) three segments  $\ell_\alpha, \ell, \ell_\omega$ , and a lexicographically sorted list of segments  $K$  with  $K = \ell'_\alpha, \dots, \ell'_\omega$ . Here  $\ell_\omega$  and  $\ell'_\omega$  are the same segment with a changed left boundary, and  $\ell_\alpha$  and  $\ell'_\alpha$  differ only in the right boundary. It is explicitly allowed that  $\ell_\alpha$  and  $\ell_\omega$  are void, with the meaning that  $\ell$  is unbounded to the left and respectively to the right. Replace the three segments by  $K$  in the active set. It is asserted that the active set forms an upper envelope after the replacement.

**Query** Given a value  $v$ , report the highest intersection of an active segment with the vertical line given by  $x = v$ .

**Subenvelope structure  $\mathcal{T}$ .** This structure allows queries on a generalization of segments, namely subenvelopes. A subenvelope is a lexicographically sorted list of line segments where neighbors have precisely one point in common. We will maintain a small upper bound on the size of an subenvelope. Again it is asserted that the segments in fact are segments from upper envelopes.

**Insert** Given a list  $L$  of segments, insert the subenvelope formed by  $L$ . Return a pointer to the newly created data structure of the subenvelope.

**Delete** Given a pointer to a subenvelope, delete that subenvelope. Return the segments of the subenvelope.

**Query** Given a value  $v$ , report the highest intersection of an inserted subenvelope with the vertical line given by  $x = v$ .

### 3.2 Dynamization

Throughout the following we assume that we know the value of  $n$ , the total number of insert operations, in advance. Standard doubling techniques justify this assumption.

Starting from the monotonic data structure presented in Sect. 2, we apply a general dynamization technique for decomposable search problems attributed to Bentley and Saxe [3]. The idea is that we divide the set of lines  $L$  into a partition  $\mathcal{C}$  based on the order the lines are inserted. More precisely every set  $C \in \mathcal{C}$  has a rank. If there are  $d$  sets of the same rank  $i$ , we merge them into one new set of rank  $i + 1$ . Sets of rank 0 have size 1. We choose the parameter  $d = \lceil \log n \rceil$ , leading to at most  $r = O(\log_d n) = O(\log n / \log \log n)$  different ranks. This is also an upper bound on the number of times a specific line can participate in the merge of  $d$  sets. Furthermore the number  $e = |\mathcal{C}|$  of sets is bounded by  $e = O(rd) = O(\log^2 n / \log \log n)$ . Every set has a deletion only structure and a set in the query structure attached.

The merge operation first deletes all the involved sets from the Query structure  $Q$ . Then it orders the lines (dual) according to their slopes, which corresponds to sorting the corresponding (primal) points according to their  $x$  coordinates. Here we exploit that the sets we are merging are already sorted in that order. We use a heap of size  $d$  to iteratively find the remaining line with smallest slope. Then we invoke the `Build` operation of the deletion only data structure, and use the reported upper envelope in an `Init set` operation of the query structure  $Q$ . We attach the returned pointer to the new set.

For an `Insert`( $\ell$ ) we create a new record for  $\ell$  that keeps the coordinates (slope and offset) and also a pointer  $p_\ell$  to the set of  $\mathcal{C}$  that currently contains  $\ell$ . Then we create a new set of size 1 and rank 0 and perform necessary merge operations. During the merge operations we update the pointers  $p_\ell$  for all lines we move.

If we want to delete a line  $\ell$  we look up the set  $C \in \mathcal{C}$  that contains  $\ell$ , and then we invoke the `Delete`( $\ell$ ) operation of the deletion only data structure from Sect. 2. This returns a list of new segments, which implicitly gives also the two neighbors of  $\ell$ . With this information we call the `Replace inside set` operation of  $Q$ .

### 3.3 Grouping

Now we implement the query structure using only a Subenvelope structure. We choose a block size parameter  $b = \lceil \log n / \log \log n \rceil$ .

The `Init set with active envelope` operation first deletes all pointers to blocks on the lines of the set. Then it groups the segments of  $K$  equally into as few as

possible blocks of size at most  $b$ . It inserts the resulting subenvelopes and stores the subenvelope pointer at every line.

The **Delete set** operation walks along the set, deleting blocks pointed to by the lines and deleting the pointers as well.

The **Replace inside an envelope** operation looks up the blocks where the three lines are stored. Then it deletes the pointed to subenvelopes, building a list  $L$  of segments that got deleted. In this list we replace  $\ell_\alpha, \ell, \ell_\omega$  by  $K$ . Then we group  $L$  optimally into blocks of size  $b$ . We insert the blocks and update the block pointers.

The query gets directly handed over. This is correct, as all active segments are in some block.

### 3.4 The Interval Tree $\mathcal{T}$ for Subenvelopes

We implement the subenvelope structure as an interval tree. The interval tree  $\mathcal{T}$  is a rooted tree. We assume to know the number  $M$  of leaves of  $\mathcal{T}$ . We choose the degree parameter  $B = \lceil \log n \rceil$ . We keep  $\mathcal{T}$  balanced by maintaining the invariants that the degree of a node is at most  $2B - 1$  and at least 2 at the root and at least  $B$  for all the other internal nodes. All leaves have the same distance to the root. A leaf  $\ell$  of  $\mathcal{T}$  stores a (possibly unbounded) interval  $I_\ell$ , its range. Every internal node  $v$  of  $\mathcal{T}$  stores its range  $I_v$ , the interval that is the (disjoint) union of the ranges of its children. To deal with a non constant degree of a node we maintain a dictionary (balanced tree) of the endpoints of the ranges of its children. For an arbitrary interval  $I$  we say that the node  $u$  of  $\mathcal{T}$  corresponds to  $I$  if the range of  $u$  contains the interval, i.e.  $I \subseteq I_u$ , and for none of the children  $v$  of  $u$  it is the case that  $I$  is contained in the range  $I_v$  of  $v$ . Note that there is always a unique node of  $\mathcal{T}$  corresponding to an interval. We can find all the intervals containing a certain point  $p$  on the path from the root node to the leaf that contains  $p$ . We assert that the range of every leaf node contains at most one endpoint of the stored intervals.

We store subenvelopes at the node in  $\mathcal{T}$  that corresponds to their interval, i.e. the extent along the  $x$ -axis. We store the segments of the subenvelope in the secondary structure at that node, i.e. as lines in a fully dynamic upper envelope structure.

The **Insert** operation creates a record that has a list of the lines forming the subenvelope, the interval, and a pointer to the node of  $\mathcal{T}$ . A pointer to this record is returned. It inserts the interval into  $\mathcal{T}$  and finds the node  $u$  in  $\mathcal{T}$  corresponding to the interval and inserts all the lines into the secondary structure  $S_u$ . It stores the returned identifiers in a list in the newly created record.

As we have the strong restriction that the range of a leaf should contain at most one endpoint of an interval stored in the tree, we might be forced to split nodes of  $\mathcal{T}$  in a bottom up fashion. Assume that node  $u$  of  $\mathcal{T}$  has too many children. Then we create a new right sibling  $v$  of  $u$  (creating a new root if  $u$  was the root) and move the right half of the children of  $u$  to  $v$ . We walk through the list of blocks being stored at  $u$ . For a block  $w$  we take  $I_w$  to decide if they should stay at  $u$ , get moved to  $v$  or moved up to the parent  $p$  of  $u$  and  $v$ . If necessary

we delete all the lines of  $w$  from the secondary structure  $S_u$  of  $u$ . If the block moves to  $v$  we insert the lines into  $S_v$ . If it moves up to  $p$ , we keep the block  $w$  “on hold”, in case that  $p$  also gets split. During this we update the pointers between the nodes of  $\mathcal{T}$  and the records of blocks.

If a subenvelope has the interval  $] -\infty, \infty[$ , it gets stored at the root of  $\mathcal{T}$ , and it cannot cause any splits. We call such a subenvelope trivial.  $M$  accounts only for non-trivial subenvelopes.

For the Delete operation we remove all the lines from the secondary structure.

For a Query operation with value  $x$ , we determine the path  $p$  in  $\mathcal{T}$  from the root to the leaf  $v$  of  $\mathcal{T}$  whose range contains  $x$ . For all nodes  $u$  on  $p$  we perform an upper envelope query for  $x$  on the secondary structure  $S_u$ . We report the topmost of the answers.

This answer is correct, because the block of the topmost segment at  $x$  is stored in one of the parents of the leaf  $v$  that contains  $x$ .

### 3.5 Analysis

**Bound on the number  $M$  of nontrivial subenvelope inserts.** We have to bound the number of operations on blocks performed within the query structure  $Q$ .

At the init operation we give every line a fractional coin that allows it to participate as a fraction  $2/b$  in a non-trivial insert operation, i.e. we need  $b/2$  such coins to pay for a non-trivial insert. Then the init operation on a set of size  $m$  costs us  $\lceil 2m/b \rceil$  non-trivial subenvelope insert operations. If the init operation gives rise to a nontrivial insert, it is paid for.

A replace operation is going to pay for 3 subenvelope deletions and 4 subenvelope insertions. If there are more blocks to be inserted, the blocks are definitely half full, and only 2 blocks on each end contain any lines that have already used their coins. The remaining block insertions can therefore be paid with coins.

Knowing that one line can only cause one replace operation and participate in  $r$  init operations, we get a total account of  $M = O(n + n \cdot r/b) = O(n + n \cdot \log n / \log \log n \cdot \log \log n / \log n) = O(n)$ .

**Bound  $s$  on the size of secondary structures in  $\mathcal{T}$ .** For every set in  $\mathcal{C}$  we have at most  $B$  subenvelopes stored at a node  $v$ . With the bounds on the size of subenvelope and on  $|\mathcal{C}|$  we get  $s = O(B \cdot b \cdot e) = O(\log^4 n)$ .

A query takes  $O(\log M + Q(s) \cdot h) = O(\log n + \log \log n \cdot \log n / \log \log n) = O(\log n)$  time.

**Work in the split operations.** Every split operation creates at least one new node. We will account on that node for all the insertions and deletions that happened during this single split.

We charge the work of moving a block during a split operation entirely to the newly created node of  $\mathcal{T}$ . For this we define the level of a node  $u$  of  $\mathcal{T}$  by stating that leaves have level 0, and that the parent of nodes on level  $i$  has

level  $i + 1$ . Now we observe that an interval stored at  $u$  has both endpoints at some leaf below  $u$ . Hence the condition of having at most one endpoint of an interval per leaf implies that we have at most  $N_i = (2B)^i$  intervals stored at a node of level  $i$ . Now let  $u$  be a node on level  $i$ . Then  $u$  was created by a split operation performed on one of its siblings  $v$ . So we know that  $v$  is also on level  $i$  and the split operation involved at most  $N_i$  intervals. Additionally we know  $e = O(rd) = O(\log^2 n / \log \log n)$  which means for large  $n$  we have  $e < B^2$  and that any node in  $\mathcal{T}$  stores at most  $e \cdot B < B^3$  intervals.

Adding these costs level by level in the tree, we get that the total number of intervals moved because of split operations is bounded by  $O((M/B)2B + (M/B^2)4B^2 + (M/B^3)B^3 + (M/B^4)B^3 + (M/B^5)B^3 + \dots) = O(M) = O(n)$ . We conclude that every subenvelope insertion causes in average constantly many moves of a subenvelope during split operations.

**Running time of the update operations in  $\mathcal{T}$ .** Given the previous paragraph, we conclude that an update operation of a nontrivial block in  $\mathcal{T}$  takes amortized  $O(\log M + b \cdot U(s))$  time for finding the correct node in  $\mathcal{T}$  and to pay for the insertions and deletions of the segments, including during split operations.

Since  $U(s) \geq \log s$ , we have  $b \cdot U(s) = \Omega(\log n / \log \log n \cdot \log \log n) = \Omega(\log n)$ , so the amortized time of a non-trivial block insert operation becomes  $O(b \cdot U(s))$ .

For trivial blocks it takes amortized  $O(U(s))$  time per segment. Note that even so the root node of  $\mathcal{T}$  is special, the upper bound  $s$  on the number of segments stored there applies as well.

**Running time of the Query structure / Fully dynamic structure.** In the init operation of the query structure we account for  $2/b$  nontrivial block insert operations for every line in the set. We already argued that this is sufficient to pay for the initial insert operation of that line (i.e. when the line appears on the upper envelope of the set we just initialized). Accounting also for the possibility of being inserted as part of a trivial block, we get a per line amortized time of  $O(U(s) + b \cdot U(s)/b) = O(U(s))$ .

Knowing that every line gets initialized at the worst  $r$  times, we get an amortized insert time for the fully dynamic data structure of  $O(r \cdot U(s)) = O(\log n / \log \log n \cdot U(s))$  as claimed in Theorem 2.

For the delete operation of the fully dynamic data structure we have to account for the delete operation in the deletion only structure, and for the replace operation in the query structure. As already argued, the replace operation has to account for a constant number of block update operations, yielding an amortized time of  $O(D(n) + b \cdot U(s)) = O(D(n) + \log n \cdot U(s) / \log \log n)$ , the bound claimed in Theorem 2.

## 4 Other Queries

With the so far explained data structure for vertical line queries we can efficiently answer a whole class of other queries on the upper envelopes. Assume the query

satisfies a so called locality property, that is for a vertical line  $q$  we can determine on which side of  $q$  the answer lies by solely examine the highest line intersecting  $q$ . Then we can use binary search to give an answer with  $O(\log n)$  vertical line queries, that is in  $O(\log^2 n)$  time. But this overhead is not always necessary. In the next section we will give an important example where the already explained data structure can be used to achieve a  $O(\log n)$  query time for a more involved query.

#### 4.1 Arbitrary Line Queries

The query we address is in the primal setting: given a point  $p$  in the plane report the two tangent lines through  $p$  touching the convex hull or state that the point is inside the convex hull. This corresponds in the dual to: given an arbitrary line, give the two intersection points of the line with the upper envelope, or “no” if no such intersection exists. The exposition here adopts the dual point of view. The important observation is, that our data structure has the same properties as the data structure in [4], the argument given there applies here as well. We only sketch the query algorithm in our setting.

We use the following fact about arbitrary line queries to navigate in the interval tree of our data structure.

**Lemma 1.** *Let  $a$  and  $b$  be to walls and  $E' \subseteq E$  a subset of lines s.t. the upper envelope of  $E'$  at  $a$  and  $b$  coincides with the upper envelope of  $E$ . Assume that an arbitrary line query for a line  $\ell$  on  $E'$  results in the right intersection point  $t$ . If  $t$  lies between  $a$  and  $b$  then also the right intersection  $T$  of  $\ell$  with  $E$  lies between  $a$  and  $b$ .*

Let  $\ell$  be the line query. The query algorithm starts at the root node of the interval tree. It performs the right intersection query on the secondary structure of the current node, updating the current answer. Then it descends to the child corresponding to the interval the current answer lies in. When it reaches a leaf, the current answer reflects the right intersection of  $\ell$  with the upper envelope of all lines.

Given that our secondary structures support line queries in  $O(\log s)$  time, we have an overall query time of  $O((\log B + \log s)h) = O(\log B \log n / \log B) = O(\log n)$ .

### 5 Applications

As a prominent example we consider the  $k$ -level of  $n$  lines, which is dually related to the  $k$ -set question on  $n$  points. For this problem Edelsbrunner and Welzl [7] gave an algorithm using the data structure of Overmars and van Leeuwen that constructs the  $k$ -level in  $O(n \cdot \log n + m \cdot \log^2 n)$  time, where  $m$  is the size of the  $k$ -level. Applying Chan’s data structure this improves to  $O(n \cdot \log n + m \cdot \log^{1+\varepsilon} n)$  time, and using our data structure this yields an improved  $O(n \cdot \log n + m \cdot \log n \cdot \log \log n)$  time bound. A randomized algorithm using expected  $O(\lambda_{t+2}(n+m) \cdot$

$\log n$ ) time has been given by Har-Peled [8], where  $\lambda_{t+2}(n+m)$  is the maximum length of a Davenport-Schinzel sequence of order  $t+2$  having  $n+m$  symbols.

Basch, Guibas and Ramkumar [4] considered a version of the segment intersection problem: given a connected family  $R$  of  $n$  red line segments and a connected family  $B$  of  $n$  blue line segments in the plane, report all intersecting pairs from  $R \times B$ . Chan [4] reported an improvement from  $O((n+m) \cdot \log^3 n)$  time using Overmars and van Leeuwen's data structure to  $O((n+m) \cdot \log^{2+\varepsilon} n)$  using Chan's data structure. We get a further improvement to  $O((n+m) \cdot \log^2 n \cdot \log \log n)$ .

## References

1. J. Basch, L. J. Guibas, and G. Ramkumar. Reporting red-blue intersections between two sets of connected line segments. In *Proc. 4th European Symposium on Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pages 302–319. Springer Verlag, Berlin, 1996.
2. M. de Berg, M. van K., M. Overmars, and O. Schwarzkopf. *Computational Geometry*. Springer-Verlag, Berlin, 1997. Algorithms and applications.
3. J. Bentley and J. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *Journal of Algorithms*, 1:301–358, 1980.
4. T. M. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. In *Proc. 40th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 92–99, 1999.
5. B. Chazelle. On the convex layers of a planar set. *IEEE Trans. Inform. Theory*, IT-31:509–517, 1985.
6. Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE, Special Issue on Computational Geometry*, 80(9):1412–1434, 1992.
7. H. Edelsbrunner and E. Welzl. Constructing belts in two-dimensional arrangements with applications. *SIAM J. Comput.*, Vol. 15, No. 1, 1986.
8. S. Har-Peled. Taking a walk in a planar arrangement. In *Proc. 40th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 100–110, 1999.
9. J. Hershberger and S. Suri. Applications of a semidynamic convex hull algorithm. *BIT*, 32:249–267, 1992.
10. J. Hershberger and S. Suri. Off-line maintenance of planar configurations. *Journal of Algorithms*, 21:453–475, 1996.
11. K. Mulmuley. Randomized multidimensional search trees: lazy balancing and dynamic shuffling. In *Proc. 32nd Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 180–196, 1991.
12. M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.
13. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer Verlag, Berlin, 1985.
14. F. P. Preparata and J. S. Vitter. A simplified technique for hidden-line elimination in terrains. *International Journal of Computational Geometry & Applications*, 3(2):167–181, 1993.
15. O. Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proc. 32nd Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 197–206, 1991.

# A Dynamic Algorithm for Maintaining Graph Partitions \*

Lyudmil G. Aleksandrov<sup>1</sup> and Hristo N. Djidjev<sup>2</sup>

<sup>1</sup> Center of Informatics, Bulgarian Academy of Sciences  
G.Bonchev Str. 25-A, 1113 Sofia, Bulgaria

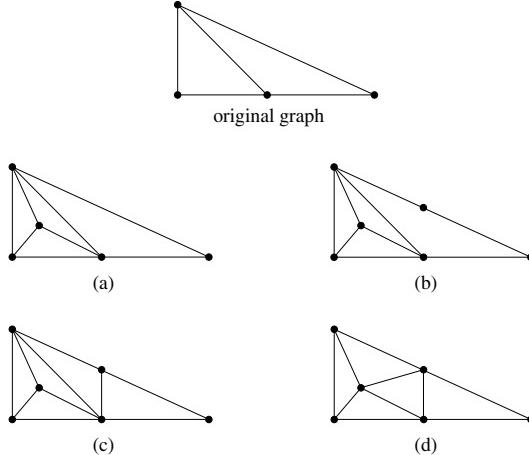
<sup>2</sup> Department of Computer Science  
University of Warwick, Coventry CV4 7AL, UK

**Abstract.** We propose an algorithm for maintaining a partition of dynamic planar graphs motivated by applications in load balancing for solving partial differential equations on a shared memory multiprocessor. We consider planar graphs of bounded face sizes that can be modified by local insertions or deletions of vertices or edges so that planarity is preserved. In our paper we describe a data structure that can be updated in  $O(\log n)$  time after any such modification of the graph, where  $n$  is the current size of the graph, and allows an almost optimal partition of a required size to be maintained. More precisely, the size of the separator is within an  $O(n^\delta)$  factor of the optimal for the class of planar graphs, where  $\delta$  is any positive constant, and can be listed in time proportional to its size. The dynamic data structure occupies  $O(n)$  space and can initially be constructed in time linear to the size of the original graph.

## 1 Introduction

Separator theorems are efficient and widely used tool for the design of efficient divide-and-conquer algorithms. Informally, a separator theorem claims that any graph from a given class can be divided into two or more parts of roughly the same size by removing a small number of vertices. The classical result of Lipton and Tarjan [15] shows that any  $n$ -vertex planar graph can be divided into components of no more than  $2n/3$  vertices by removing a set of no more than  $\sqrt{8n}$  vertices. Moreover, such a separator can be found in  $O(n)$  time. Other interesting results include separator theorems for the class of graphs of bounded genus [6,8,1], the class of graphs of excluded minor [2], and classes of geometric graphs [17]. Separator theorems have applications in solving efficiently large sparse systems of linear equations [14,9], for developing algorithms for VLSI layout design [4,12], for shortest path problems [7], in parallel computing [10], and in computational complexity [16].

\* This work was partially supported by the EPA grant R82-5207-01-0, EPSRC grant GR/M60750, and RTDF grant 98/99-0140. A two-page abstract of this work appeared in the proceedings of CCCG'98.



**Fig. 1.** Mesh refinement operations: (a) adding a point inside a face; (b) adding a point onto an edge; (c) adding an edge; (d) edge flipping.

A class of problems for whose solutions separator theorems are especially well suited is data partitioning and load balancing for parallel computing. The modern high performance computing systems have large number of processors and their memory is distributed among the processors. In order to achieve high efficiency and speed when using such computers, the data has to be allocated among the processors so that the computational load is even and the need for communication is minimized. Since this mapping problem is NP-hard, several approaches have been tried to find good approximate solutions. Popular partitioning techniques include Kernighan-Lin's local search algorithm [13], recursive spectral bisection [18], simulated annealing [11], and graph separators [19]. The advantage of the graph-separator approach is that it works well for unstructured meshes with good topology and that it gives a good guaranteed worst-case performance.

In many time-dependent applications, after the initial partition, the data might need to be modified and then reallocated to the processors. Thus the problem for dynamic load balancing arises. For instance, in solving partial differential equations, the mesh might need to be modified (refined or coarsened) after every few time steps. For the case of triangular or bounded face size planar meshes discussed in this paper refinement types of modifications include adding a new point or edge inside an existing face, adding a new point onto an existing edge, edge or face flipping, and others [Figure 1]. Efficient algorithms should use the existing partition in order to compute the new one faster and with a small number of data reallocations.

Unfortunately, there are no known deterministic algorithms that can recompute efficiently the separator after modification of the graph. The existing algorithms need to compute a separator of the new graph from scratch, without

using any information from the previous separator. Armon and Reif constructed a dynamic separator algorithm for planar graphs [3], but their algorithm is probabilistic (works in  $O(\log^3 n)$  expected time per update) and it needs as an input the so called sphere packing representation [5] of the input planar graph. Although such a sphere representation is known to always exist, it is currently not known whether it is computable in polynomial time.

The main difficulty for designing a fast deterministic algorithm for maintaining partitions of dynamic planar graphs is related to the fact that most known algorithms for constructing separators for static graphs use a breadth-first search as essential step of the computation. There are no known algorithms that can dynamically maintain a breadth-first tree of a planar graph in polylogarithmic time.

In this paper we develop an algorithm that avoids recomputation of breadth-first trees. Our approach is based on a representation of the current graph as a hierarchy of partitions of different grades for that graph. By handling the modification at an appropriate partition level, we can achieve rebalancing by reallocating a small number of faces. Specifically, we prove that our representation allows a separator decomposition to be recomputed in  $O(\log n)$  time after a local insertion or deletion of a vertex or edge that preserves planarity, biconnectivity, and bounded face sizes, where  $n$  is the current number of faces of the graph. The total initial partition and the data structure can be computed in  $O(n)$  time by using  $O(n)$  space.

The paper is organized as follows. In Section 2, we introduce the notation and outline our approach. In Section 3, we specify the requirements to the partitions at each level of the hierarchy and describe efficient algorithm for their construction. The data structure describing the partitions at different levels of the hierarchy and its dynamic maintenance are discussed in Section 4. Finally, in Section 5 we formulate our main theorem and comment the results and their implications.

## 2 Preliminaries and Algorithm Outline

### *Embeddings and Regions*

A graph is *planar* if it can be embedded in the plane so that no two edges intersect except possibly at a shared endpoint. A graph  $G$  already embedded in the plane is called a *plane* graph. *Faces* of  $G$  are the maximal connected regions into which the embedding of  $G$  divides the plane. There is exactly one infinite face called the *outer* face of the embedding, all other faces are called *internal*. A face can be identified by the cycle of edges on its boundary. By  $V(G)$ ,  $E(G)$ , and  $F(G)$  we will denote the sets of vertices, edges, and faces of  $G$ , respectively.

Any set of faces is called a *region* of  $G$ . The subgraph that consists of the vertices and the edges incident with the faces of a region  $R$  is called a subgraph *induced* by  $R$  and will be denoted by  $G(R)$ . A region  $R$  is *connected* if the dual

graph of  $G(R)$  is biconnected. The maximal connected subregions of  $R$  are called *connected components* of  $R$ .

An edge  $e$  of  $G(R)$  is called an *inner edge* of  $R$  if both faces incident to  $e$  belong to  $R$ . Otherwise  $e$  is called a *boundary edge*. The boundary  $\partial R$  of a region  $R$  is the subgraph induced by the boundary edges of  $R$ .

A *partition*  $\mathcal{R}$  of  $F(G)$  is called any set of regions  $\{R_1, \dots, R_r\}$  such that each face of  $G$  belongs to exactly one region of  $\mathcal{R}$ . A partition  $\mathcal{R}$  is called *connected* if all of its regions are connected and it is called *weakly connected* if it is either connected or each region has at most two neighboring regions. The *boundary*  $\partial\mathcal{R}$  of  $\mathcal{R}$  (called also *boundary graph* of  $\mathcal{R}$ ) is the union of the boundaries  $\partial R_i$ ,  $i = 1, \dots, r$ , of its regions.

In the typical case  $\partial\mathcal{R}$  may contain long paths of degree 2 vertices. In order to save time and space when dealing with such boundaries, we define a *compressed boundary*  $CB(\mathcal{R})$  of  $\partial\mathcal{R}$  to be the plane graph resulting after the contraction of any maximal simple path of degree 2 vertices in  $\partial\mathcal{R}$  to a single edge and any simple cycle to a triangle.  $CB(\mathcal{R})$  is a graph with  $|\mathcal{R}|$  internal faces corresponding to the regions of  $\mathcal{R}$ .

**Definition 21** Let  $G$  be an  $n$ -face plane graph and  $\varepsilon > 0$ . The partition  $\mathcal{R}$  is said to be an  $\varepsilon$ -partition of  $G$  if no region of  $\mathcal{R}$  has more than  $\varepsilon n$  faces.

We will make use of the following generalization of a result from [1].

**Theorem 1** Let  $G$  be a plane graph with  $n$  faces whose maximal size is  $d$  and let  $h \geq d^2$  be an integer. Then there exists a weakly connected  $h/n$ -partition  $\mathcal{R} = \{R_1, \dots, R_r\}$  of  $G$  that satisfies the conditions

$$(i) |\partial R_i| \leq \sqrt{h}; \quad (1)$$

$$(ii) |\mathcal{R}| \leq cn/h, \quad (2)$$

where  $c > 1$  is a constant.

In this paper we present an algorithm for maintaining a hierarchy of  $\varepsilon$ -partitions of a dynamic planar graph  $G$ . This algorithm can be used to solve the load balancing problem, since for any integer  $p > 1$  one can pick an  $\varepsilon$ -partition for  $\varepsilon \approx 1/p$  and then distribute the regions between the processors using a greedy method so that the sum of the sizes of all regions assigned to a processor does not exceed  $n/p + \varepsilon n \leq 2n/p$ . If better balancing is required, we just need to reduce the value of  $\varepsilon$  accordingly.

Our algorithms will handle the following update and query operations. We assume that the original graph is plane with face size bounded by some constant  $d$  and that no operation violates planarity or creates a face with size greater than  $d$ .

- *insert\_vertex(v,e)*: Adds a new vertex  $v$  and replace edge  $e = (u,w)$  by two edges  $(u,v)$  and  $(v,w)$ .

- *delete\_vertex(v)*: Deletes vertex  $v$  of degree two and its incident edges  $(u, v)$  and  $(v, w)$  and adds edge  $(u, w)$ .
- *insert\_edge( $u, w, F$ )*: Adds a new edge  $(u, w)$  inside the face  $F$ . Assumes  $u$  and  $w$  are non-adjacent vertices on  $F$ .
- *delete\_edge( $e$ )*: Deletes edge  $e = (u, w)$ . Assumes the degrees of  $u$  and  $w$  are greater than two.
- *list\_separator( $\varepsilon$ )*: Given any  $\varepsilon > 0$ , lists an  $\varepsilon n$ -separator of the current graph.

Note that these update operations are powerful enough to allow any plane graph to be transformed into any other plane graph using a sequence of update operations.

## Our Approach

We need to maintain a representation of the graph  $G$  that will support fast separator queries, where each query will ask for an  $\varepsilon$ -partition of the current graph. For that purpose we maintain a balanced tree called a *partition tree* that represents a hierarchy of partitions of  $G$ . The partition corresponding to the root defines the coarsest partition of  $G$  into at most  $h$  regions of roughly the same size, where  $h$  will be an appropriately chosen constant. Each subsequent level defines a finer partition, with the leaves corresponding to single faces. Thus, the levels of the partitions tree form a hierarchy of partitions of  $G$ . In particular, the leaves of  $T$ , on level 0, represent the faces of  $G$ , the nodes at the level 1 represent the regions of an  $h/n$ -partition  $\mathcal{R}^1$ , the nodes at level 2 represent regions of an  $h/n$ -partition of the graph  $CB(\mathcal{R}^1)$ , and so on.

When a local change is applied on the graph  $G$ , it can affect only nodes that are ancestors of the face where the change occurs. The algorithm maintains a number of local invariants that guarantee the validity of the partitions and the small height of the tree. The algorithm locates the lowest level of  $T$ , where some of these invariants is violated (if any). Suppose that at node  $N$  any of these invariants is violated. The algorithm considers the subtree consisting of  $N$  plus its children and grandchildren and redefines the children of  $N$  by finding a new partition of the graph induced by the grandchildren of  $N$ . This subtree has  $O(1)$  size and thus it can be rebalanced in  $O(1)$  time so that all invariants hold. Since the same procedure might need to be applied to all ancestor nodes from  $N$  to the root of  $T$ , the total update time is proportional to the height of  $T$ , which will be shown to be  $O(\log n)$ .

## 3 P–Tree Data Structure

In this subsection we define and study the properties of a data structure, called a *P–tree*, for describing partitions of a plane graph  $G$  and show that the structure can be constructed in linear time. For the initial construction of the tree, we will define a sequence (hierarchy) of partitions whose elements will be then stored at the nodes of the tree.

More precisely, let  $G$  be a plane graph with  $n$  faces each of size not exceeding  $d$ . Let  $h$  be an integer constant such that  $h \geq \max(d^2, 2c)$ , where  $c$  is the constant

from Theorem II. By applying Theorem II iteratively, we will construct a sequence of graphs and their  $h$ -partitions

$$G = G^{(0)}, \mathcal{R}^{(1)}, G^{(1)}, \dots, \mathcal{R}^{(l)}, G^{(l)}$$

called a *GR-sequence*, so that the graph  $G^{(0)}$  is the original graph  $G$  and the region partition  $\mathcal{R}^{(l)}$  consists of a single region of no more than  $h$  faces.

### **Constructing the GR-Sequence**

Assuming that  $G^{(i-1)}$  has already been constructed for some  $i > 1$  and that it has  $n_{i-1} > h$  faces with each face boundary of at most  $\sqrt{h}$  edges, we can construct  $G^{(i)}$  from  $G^{(i-1)}$  by applying the following procedure. We let  $G^{(0)} = G$ .

#### **Algorithm GR-SEQUENCE**

Step 1: Apply Theorem II with a parameter  $h$  on  $G^{(i-1)}$  to find an  $h/n_{i-1}$ -partition  $\mathcal{R}^{(i)}$  of  $G^{(i-1)}$  such that  $|\mathcal{R}^{(i)}| \leq cn_{i-1}/h$ .

Step 2: For each region  $R \in \mathcal{R}^{(i)}$  construct the subgraph  $G^{i-1}(R)$  of  $G^{i-1}$  induced by  $R$  and find its boundary  $\partial R$ .

Step 3: If  $\mathcal{R}^{(i)}$  is a connected partition, then  $G^{(i)}$  will just be the compressed boundary of  $\mathcal{R}^{(i)}$ . In the general case, replace each region of  $\mathcal{R}^{(i)}$  by a single face of  $G^{(i)}$  as follows.

3.1. Compute the compressed boundary graph  $CB = CB(\mathcal{R}^{(i)})$  of  $\mathcal{R}^{(i)}$ . For each new edge  $e$  defined during the compression store the list of edges of the path corresponding to  $e$ .

3.2. For any non-connected region  $R_{nc}$  of  $\mathcal{R}^{(i)}$  merge the faces in  $CB$  corresponding to the connected components of  $R_{nc}$  into one of those faces (arbitrarily selected).

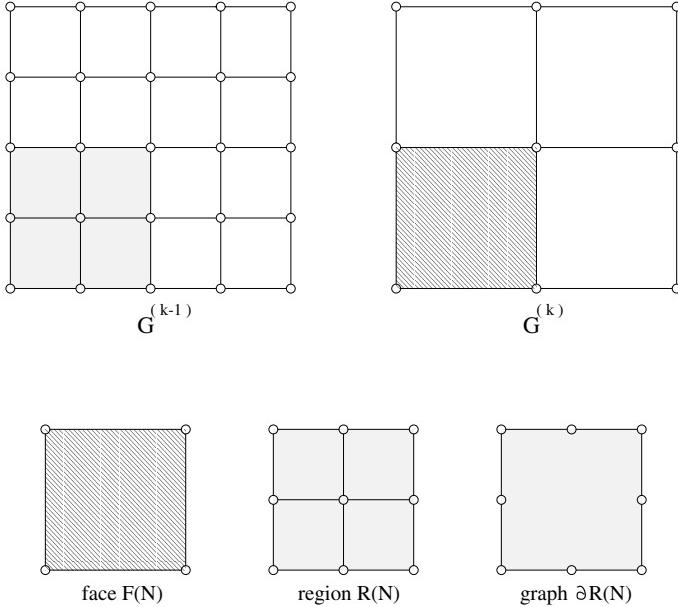
Note that the above algorithm constructs  $G^{(i)}$  together with its embedding in the plane and its faces correspond to the regions of  $\mathcal{R}^{(i)}$ . According to Theorem II  $G^{(i)}$  has no more than  $cn_{i-1}/h$  faces and the size of each face does not exceed  $\sqrt{h}$ . Thus the same algorithm can be used for constructing  $G^{(i+1)}$  assuming  $G^{(i)}$  has at least  $h$  faces.

Since the parameter  $h$  was chosen to be at least  $2c$ , the graph  $G^{(l)}$  will be obtained in no more than  $\log n + 1$  iterations. Iteration  $k$  takes time proportional to the size of  $G^{(k-1)}$  and thus the total time for constructing the *GR*-sequence will be

$$\sum_{k=0}^{l-1} O((c/h)^k n) = O(n).$$

### **Constructing the P-Tree**

Let  $G$  be a plane graph for which a *GR*-sequence  $S$  has already been constructed. A *P-tree* for  $G$  with respect to  $S$  is a data structure whose elements are associated with a rooted tree  $T_h(G)$  with  $l+1$  levels. The  $k$ -th level of  $T_h(G)$



**Fig. 2.** Information associated with node  $N$  at level  $k$  of  $T_h(G)$ .  $N$  has 4 children corresponding to the 4 faces of  $R(N)$ . The relevant subgraphs of  $G^{(k-1)}$  and  $G^{(k)}$  are shown. Each face of  $G^{(k-1)}$  corresponds to a region of  $G$ .

for  $k = 1, \dots, l$  contains the information gathered during the  $k$ -th iteration of the construction of  $S$ , namely, the partition  $\mathcal{R}^{(k)}$ , its boundary  $\partial\mathcal{R}^{(k)}$ , and the graph  $G^{(k)}$ . The nodes at the  $k$ -th level of the P-tree correspond to the regions of  $\mathcal{R}^{(k)}$  (or, equivalently, to the faces of  $G^{(k)}$ ). The leaves of  $T_h(G)$  are at level 0 and each leaf corresponds to a face of the original graph  $G = G^{(0)}$ . The root node is at level  $l$  and contains the graph  $G^{(l)}$ .

With each node  $N$  on level  $k > 0$  we associate the following information (Figure 2):

- (i)  $F(N)$  – the face of  $G^{(k)}$  corresponding to  $N$ ;
- (ii)  $\mathcal{R}(N)$  – the region of  $G^{(k-1)}$  corresponding to  $F(N)$ ; and
- (iii)  $\partial\mathcal{R}(N)$  – the boundary graph of  $\mathcal{R}(N)$ .

If  $N$  is a node on level 0, then  $F(N)$  is defined as in (i) above and  $\mathcal{R}(N) = \partial\mathcal{R}(N) = F(N)$ . Let  $G(N)$  denote the graph  $G(\mathcal{R}(N))$ .

Note that the amount of data associated with any node  $N$  of  $T$  is proportional to the size of  $G(N)$ . Since  $G(N)$  is a planar graph with no more than  $h$  faces and each face has size at most  $\sqrt{h}$ , the size of  $G(N)$  is  $O(h^{3/2})$  (which is a constant since  $h = O(1)$ ).

The edges of  $T_h(G)$  (called *links* hereafter to be distinguishable from the edges of  $G^{(k)}$ ) connect certain pairs of nodes on consecutive levels. More precisely, there is a link between a node  $N$  from level  $k - 1$  corresponding to a region  $R$ , and

a node  $N'$  from level  $k$  corresponding to a region  $R'$ , iff  $R$  is transformed by Algorithm GR-SEQUENCE into a face of  $R'$ . By Theorem 1, each node of  $T_h(G)$  has at most  $h$  children.

Note that any edge of  $G^{(k-1)}$  appears in either three or four nodes of the tree  $T_h(G)$ . It appears twice on the  $(k-1)$ -th level at the nodes that correspond to its two incident faces and also at one or two nodes on the  $k$ -th level, depending on whether this edge is entirely inside some of the regions of  $\mathcal{R}^k$ , or lies between two such regions. We assume that there are pointers between any edge and its occurrences on the lower level. These pointers define an *edge forest*  $EF$  on the set of edges of  $G^{(k)}$  for  $k = 0, \dots, l$  that is consistent with  $T_h(G)$ . Namely, all edges on path that is compressed to an edge  $e$  are descendants of  $e$ . Thus, the edges of the graph  $G^{(k)}$  for  $k > 1$  have descendants that are edges of  $G^{(k-1)}$ . On the other hand, an edge  $e'$  from level  $k-1$  has an ancestor if and only if  $e'$  is on the boundary of some of the regions of  $\partial R^k$ .

In accordance with  $EF$ , we define *weights*  $wt(\cdot)$  on the edges of  $G^{(k)}$  as the number of their descendants in  $F$  at level 0. The weight of the boundary of a region or a face is defined as the sum of the weights of the edges on that boundary.

**Definition 31** A  $P$ -tree node  $N$  is called balanced, if all of the following three conditions are satisfied.

- (B1)  $N$  and each child of  $N$  have no more than  $h$  children each.
- (B2) The ratio between the number of the children and the number of the grandchildren of  $N$  does not exceed  $c/h$ , where  $c$  is the constant from Theorem 1.
- (B3) For any child  $N_1$  of  $N$   $wt(\partial F(N_1)) \leq dh^{(k-1)/2}$ , where  $k$  is the level of  $N$ .

A  $P$ -tree is called balanced if all its non-root nodes are balanced.

By Algorithm GR-SEQUENCE and Theorem 1 we have the following.

**Lemma 31** If  $G$  is a plane graph with  $n$  faces, then the algorithm from this section constructs in  $O(n)$  time a balanced  $P$ -tree for  $G$ .

*Proof.* Since Step 1 of Algorithm GR-SEQUENCE constructs an  $h/n_{i-1}$ -partition of  $G^{(i-1)}$ , any node of the tree has at most  $h$  children, implying (B1). Property (B2) follows from conditions (1) and (2) of Theorem 1. Finally, Property (B3) follows from condition (1) of Theorem 1, the assumption that the maximum face size of the original graph is no more than  $d$ , and an induction on  $k$ .

We will maintain our  $P$ -tree balanced.

### 3.1 Update and Query Operations

Next we describe the basic operation on  $P$ -trees used to maintain the balance property of nodes. It is possible that more than one node is unbalanced at a

time, but all unbalanced nodes belong to a single simple path from certain node to the root of the tree.

## Balancing the Tree

We will first describe the basic algorithm for maintaining the balance of the  $P$ -tree, called *Fix-Tree*.

Algorithm FIX-TREE starts at an unbalanced node  $N$  on a lowest possible level and makes the subtree consisting of  $N$  and all its descendants balanced in constant time. Then the same operation is applied on the parent of  $N$ , and so on, until the root is processed.

Let  $N$  be a node at level  $k$ ,  $2 \leq k \leq l + 1$ , that is unbalanced, but all its proper descendants are balanced. Denote the subgraph of  $G^{(k-2)}$  induced by the faces of the grandchildren of  $N$  by  $G^{(k-2)}(N)$ . Repartition  $G^{(k-2)}(N)$  using the algorithm from Theorem I and construct again the portion of the tree (with height two) rooted at  $N$ , its children, and its grandchildren using algorithm similar to the one for construction of  $P$ -trees. The time required to fix  $N$  will be proportional to the number of faces of  $G^{(k-2)}(N)$ , which is  $O(h^2)$  by of Property (B1) of Definition B1. Since the height of the  $P$ -tree is  $O(\log n)$ , balancing the tree will take  $O(\log n)$  time.

## Implementation of the Update Operations

We will describe and analyze the implementation of the four update operations described in Section 2. Our assumption is that the current graph  $G$  is a connected plane graph with no vertices of degree 1 and no face size exceeding  $d$  and that the update operations preserve these properties. By  $T$  we denote the current  $P$ -tree of  $G$ .

*insert\_vertex( $v, e$ )* The operation asks that edge  $e = (u, w)$  be replaced by edges  $e_1 = (u, v)$  and  $e_2 = (v, w)$ .

Let  $F_1$  and  $F_2$  be the two faces of  $G$  incident to  $e$  and let  $N_1$  and  $N_2$  be the corresponding leaf nodes in  $T$ . First, update at nodes  $N_1$  and  $N_2$  the corresponding descriptions of  $F_1$  and  $F_2$  by deleting  $e$  and inserting  $e_1$  and  $e_2$  into the corresponding doubly linked lists. Update the weights of all edges that were ancestors of  $e$  in the edge forest  $EF$  associated with  $T$ . Since these updates can change the weights of the faces associated with the ancestors of  $N_1$  and  $N_2$ , we need to apply *Fix\_Tree* algorithm on the lowest unbalanced ancestors of each of  $N_1$  and  $N_2$ .

*delete\_vertex( $v$ )* This operation deletes a vertex  $v$  of degree two, thereby changing the common boundary of the two faces incident to  $v$ . Hence, *delete\_vertex* can be implemented in a similar way as *insert\_vertex* by changing the information associated with the two leaf nodes representing the faces incident to  $v$  and updating the weights of  $O(1)$  edges per each level of the edge-forest associated with  $T$ .

*insert\_edge( $u, w, F$ )* Given two non-adjacent edges  $u$  and  $w$  on the same face  $F$  of  $G$ , *insert\_edge* adds a new edge  $e = (u, w)$  inside  $F$ .

Denote by  $F_1$  and  $F_2$  the faces into which  $e$  splits  $F$ . Delete the leaf node  $N$  that corresponds to  $F$  and create two new nodes  $N_1$  and  $N_2$  representing the new faces. Make  $N_1$  and  $N_2$  children of the parent  $PN$  of  $N$ . Next, update the region subgraph stored at  $PN$  by adding to  $\mathcal{R}(PN)$  the edge  $e$  inside face  $F$ . Finally, execute algorithm *Fix\_Tree* on the lowest unbalanced ancestor of  $PN$ .

*delete\_edge( $e$ )* This operation deletes edge  $e = (u, w)$ , assuming the degrees of  $u$  and  $w$  are greater than two.

Let  $F_1$  and  $F_2$  be the two faces of  $G$  incident to  $e$  and let  $N_1$  and  $N_2$  be the corresponding nodes in  $T$ . Let  $F$  be the face that results after the deletion of  $e$  and the merge of  $F_1$  and  $F_2$ . In order to implement that operation, we delete  $N_1$  and  $N_2$  and create a new leaf node  $N$  representing the face  $F$ .

Assume that  $N_1$  and  $N_2$  have different parents, say  $PN_1$  and  $PN_2$ . In this case we delete  $N_2$  and replace  $N_1$  with the new node  $N$ . As a result, the number of children of  $PN_2$  is reduced by one and one of the children of  $PN_1$  represents a larger region. We make the corresponding changes in the parents  $PN_1$  and  $PN_2$ . These include updates in the structures  $F(PN_i)$ ,  $\mathcal{R}(PN_i)$ , and  $\partial\mathcal{R}(PN_i)$ ,  $i = 1, 2$ . In particular,  $e$  is deleted from both faces  $F(PN_1)$  and  $F(PN_2)$  and all edges of  $F_2$ . We continue to make similar changes on the ancestors of  $F(PN_1)$  and  $F(PN_2)$  until the nearest common ancestor of  $N_1$  and  $N_2$  is reached.

Finally, we run the Algorithm *Fix\_Tree* on the parents of  $N_1$  and  $N_2$ .

If in result of some update operation the degree of the root  $R$  becomes one, then we cut  $R$ . In case the degree of  $R$  becomes greater than  $h+1$ , then we define a new vertex  $R'$  to be a parent of  $R$  and run Algorithm *Fix\_Tree* on vertex  $R'$ .

Note that any of the update operations described above changes information at only constant number of nodes at any level of  $T$  and that we applied *Fix\_Tree* on only one or two nodes of  $T$ . If all nodes of  $T$  are balanced, then by Condition (B2) the height of  $T$  is at most  $\log_{n/c} n \leq \log n$ , since we have chosen  $h \geq 2c$ . Hence the time complexity of Algorithm *Fix\_Tree* is also  $O(\log n)$ . Thus we have the following lemma.

**Lemma 32** *A balanced P-tree representing a planar connected graph  $G$  with no vertex of degree one can be maintained subject to the operations *insert\_vertex*, *delete\_vertex*, *insert\_edge*, and *delete\_edge* in  $O(\log n)$  time per operation, assuming the maximum face size never exceeds a constant  $d$ .*

## Extraction of an $\varepsilon$ -Partition

Recall that any node  $N_i$  at the  $k$ -th level represents a face  $F(N_i)$  of  $G^{(k)}$ . Similarly, the children of  $N_i$  correspond to the faces of  $\mathcal{R}(N_i)$  (the region of  $G^{(k-1)}$  associated with  $F(N_i)$ ), and so on. Therefore,  $N_i$  defines a region in the graph  $G = G^{(0)}$  represented by the leaves that are descendants of  $N_i$ . We denote that region of  $G$  by  $\bar{R}(N_i)$ . Let  $\mathcal{R}(G, k)$  denotes the partition  $\{\bar{R}(N_1), \dots, \bar{R}(N_s)\}$  of  $G$ , where  $\{N_1, \dots, N_s\}$  is the set of all nodes on level  $k$ .

**Lemma 33** Let  $1 \leq k \leq l$  and  $\varepsilon_k = h^k/n$ , where  $n$  is the number of faces in  $G$ . Then the partition  $\mathcal{R}(G, k)$  is an  $\varepsilon_k$ -partition of  $G$  with boundary of size not exceeding  $d\sqrt{\frac{n^{1+\log c/\log(h/c)}}{\varepsilon_k}}$ .

*Proof.* Clearly,  $\mathcal{R}(G, k)$  is a partition of  $G$  since any face of  $G$  (which is a leaf of the  $P$ -tree of  $G$ ) belongs to exactly one of the regions  $\bar{R}(N_i)$ . For the size of the regions of that partition  $\mathcal{R}(G, k)$  we have

$$|\bar{R}(N_i)| \leq h^k = \varepsilon_k n, \text{ for } i = 1, \dots, s,$$

since the maximum degree of  $T$  is no more than  $h + 1$ . For the size of the boundaries  $\partial\bar{R}(N_i)$  we have

$$|\partial\bar{R}(N_i)| = \text{wt}(F(N_i)) \leq dh^{k/2},$$

according to Condition (B2) of Definition 31. The number of nodes at level  $k$  is at most  $h^{l-k}$ , where  $l + 1$  is the number of the levels in  $T$ . Therefore, the total weight of the boundary of  $\mathcal{R}(G, k)$  is

$$\begin{aligned} \text{wt}(\mathcal{R}(G, k)) &\leq dh^{k/2}h^{l-k} = dh^{l-k/2} \leq d\frac{h^{\log n/\log(h/c)}}{h^{\frac{k}{2}}} \\ &= d2^{\frac{\log n \log h}{\log(h/c)}} / \sqrt{\varepsilon_k} = d\sqrt{n^{1+\log c/\log(h/c)}/\varepsilon_k}. \end{aligned}$$

Now, in order to process a query asking for an  $\varepsilon$ -partition of  $G$  for some  $\varepsilon \in (0, 1)$ , we determine the level  $k$  for which  $h^{k-1} < \varepsilon n \leq h^k$  and apply Lemma 33. Let us estimate the time necessary to list the boundary of the resulting partition. Recall that  $F(N)$  is a compressed image of the boundary  $\partial\mathcal{R}(N)$  of the region  $\mathcal{R}(N)$  of  $G^{(k-1)}$ . Each edge of  $F(N)$  represents a path on  $\partial\mathcal{R}(N)$ . Each edge of  $\partial\mathcal{R}(N)$  itself represents a path in  $G^{(k-2)}$  (if  $k \geq 2$ ), and so on. Therefore, each edge of  $F(N)$  can be traced down to a path in the original graph  $G = G^{(0)}$ . The boundary  $\partial\bar{R}(N)$  of the region  $\bar{R}(N)$  can be extracted from the  $P$ -tree in time  $O(\partial\bar{R}(N))$ . Thus the boundary of the partition can be listed in time proportional to its size, which is  $O(\sqrt{n/\varepsilon})$ . Thus we have the following lemma.

**Lemma 34** The  $k$ -th level of a  $P$ -tree represents a  $h^k$ -partition of  $G$  whose boundary  $B$  can be listed in  $O(|B|)$  time.

We summarize the above results in the following theorem.

**Theorem 2** Let  $G$  be a plane graph with  $n$  faces. Then for any  $\delta > 0$  a data structure exists that

- (i) can be constructed in  $O(n)$  time;
- (ii) supports operations `insert_vertex`, `delete_vertex`, `insert_edge` and `delete_edge` in  $O(\log n)$  time assuming the maximum face size of any intermediate graph is  $O(1)$ .
- (iii) supports operation `list_separator` ( $\varepsilon$ ) for any  $\varepsilon > 0$  in time proportional to the separator's size, which is bounded by  $O(\sqrt{n^{1+\delta}/\varepsilon})$ .

*Proof.* Follows from Lemmas 31–34 for  $h = c^{1/\delta}$ .

## References

1. Lyudmil Aleksandrov and Hristo N. Djidjev. Linear algorithms for partitioning embedded graphs of bounded genus. *SIAM Journal on Discrete Mathematics*, 9:129–150, 1996.
2. Noga Alon, Paul Seymour, and Robin Thomas. A separator theorem for graphs with an excluded minor and its applications. *Proceedings of the 22nd Symp. on Theory of Computing*, pages 293–299, 1990.
3. D. Armon and J. Reif. A dynamic separator algorithm. In *Proc. 3rd Worksh. Algorithms and Data Structures*, pages 107–118. Lecture Notes in Computer Science 709, Springer-Verlag, Berlin, 1993.
4. S.N. Bhatt and F.T. Leighton. A framework for solving VLSI graph layout problems. *Journal of Computer and System Sciences*, 28:300–343, 1984.
5. J.H. Conway and N.J.A. Sloane. *Sphere Packings, Lattices and Groups*. Springer-Verlag, 1988.
6. Hristo N. Djidjev. A separator theorem. *Compt. rend. Acad. bulg. Sci.*, 34:643–645, 1981.
7. G.N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16:1004–1022, 1987.
8. John R. Gilbert, Joan P. Hutchinson, and Robert E. Tarjan. A separator theorem for graphs of bounded genus. *J. Algorithms*, 5:391–407, 1984.
9. John R. Gilbert and Robert E. Tarjan. The analysis of a nested dissection algorithm. *Numerische Mathematik*, 50:377–404, 1987.
10. Michael T. Goodrich. Planar separators and parallel polygon triangulation. *Proceedings of 24th Symp. on Theory of Computing*, pages 507–516, 1992.
11. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, pages 671–680, 1983.
12. C.E. Leiserson. Area efficient VLSI computation. In *Foundations of Computing*. MIT Press, Cambridge, MA, 1983.
13. B.W. Kernighan & S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell Systems Technical Journal*, 49:291–308, 1970.
14. Richard J. Lipton, D. J. Rose, and Robert E. Tarjan. Generalized nested dissection. *SIAM J. Numer. Anal.*, 16:346–358, 1979.
15. Richard J. Lipton and Robert E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36:177–189, 1979.
16. Richard J. Lipton and Robert E. Tarjan. Applications of a planar separator theorem. *SIAM Journal on Computing*, 9:615–627, 1980.
17. Gary L. Miller, Shang-Hua Teng, and Stephen A. Vavasis. A unified geometric approach to graph separators. *Proceedings of the 32nd FOCS*, pages 538–547, 1991.
18. A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, July 1990.
19. Eric J. Schwabe, Guy E. Blelloch, Anja Feldmann, Omar Ghattas, John R. Gilbert, Gary L. Miller, David R. O'Hallaron, Jonathan R. Shewchuk, and Shang-Hua Teng. A Separator-Based Framework for Automated Partitioning and Mapping of Parallel Algorithms for Numerical Solution of PDEs. In *Proceedings of the First Annual Summer Institute on Issues and Obstacles in the Practical Implementation of Parallel Algorithms and the Use of Parallel Machines in Parallel Computation (DAGS/PC '92)*, pages 48–62. Dartmouth Institute for Advanced Graduate Studies, June 1992.

# Data Structures for Maintaining Set Partitions

## (Extended Abstract)

Michael A. Bender, Saurabh Sethia, and Steven Skiena

Department of Computer Science, State University of New York at Stony Brook,  
Stony Brook, NY 11794-4400 USA, {bender|saurabh|skiena}@cs.sunysb.edu

## 1 Introduction

Each test or *feature* in a classification system defines a *set partition* on a class of objects. Adding new features refines the classification, whereas deleting features may result in merging previously distinguished classes. As an illustration, consider the set of automobile types { VW Beetle, Toyota, Lexus, Cadillac }. The feature *size* partitions the cars into sets of small and large cars, {{ VW Beetle, Toyota}, { Lexus, Cadillac }}. The feature *domestic-origin* partitions the cars into {{ VW Beetle, Toyota, Lexus }, { Cadillac }}. The feature *ugly-shape* distinguishes { VW Beetle, Cadillac } from { Toyota, Lexus }. Incorporating both *size* and *origin* induces the refined partition {{ VW Beetle, Toyota}, { Lexus }, { Cadillac }}, whereas the union of all three features completely distinguishes the types of cars. In fact, *size* and *ugly-shape* are sufficient for complete identification, so *domestic-origin* could be deleted from the set of features without affecting the induced partition.

Efficiently maintaining the partition induced by a set of features is an important problem in building decision tree classifiers. For example, in building an optical character recognition (OCR) system [15][16] based on point-probe decision trees [1], each of the 1500-plus pixels in each character-sized window of the image may be evaluated as a possible feature. An important goal is to find a small, robust set of probe points sufficient to distinguish among the 70-plus characters in a font, a process that may require repeatedly inserting and deleting features to see the impact on the final classification.

In this paper, we introduce techniques to speed up this process of feature identification. We propose a series of data structures for maintaining a collection of set partitions on elements  $U = \{1, \dots, n\}$ . The data structures efficiently support the following three operations:

- *Insert( $P, S$ )* – add a new partition  $P$  to the set of partitions  $S$ .
- *Delete( $P, S$ )* – delete existing partition  $P$  from the set of partitions  $S$ .
- *Report( $S$ )* – report the set partition of  $U$  induced by the set of partitions in  $S$ .

*Previous Work.* A variety of data structures for sets and set partitions are known, including dictionaries and bit vectors, but these are not directly applicable to our problem. The primary difficulty of our problem lies in the fact that deleting

a set partition may or may not result in the merger of two parts of the current induced partition, depending upon which other set partitions are included in the data structure. Union-find data structures [17] provide some support for merging disjoint subsets (as occurs on deleting a partition), but do not permit us to break up subsets (as occurs on adding a partition).

Partition refinement techniques are used in a variety of algorithms, notably minimizing deterministic finite automatas [12] and its generalizations [14]. Habib, et.al. [11] demonstrate that partition refinement can lead to simple and efficient algorithms for graphs, strings, and matrices – although none of these operations involves deleting arbitrary set partitions.

Yellin [20] efficiently supports a variety of subset testing operations (insert/delete elements, create subsets, subset and intersection queries) in  $O(n^{1/2} \log n)$  time per operation, but it is not clear how to use these operations to improve even the naive bounds for our problem. Further, near matching lower bounds are known on the complexity of any data structure that supports these operations [6]. We have recently learned of a data structure for maintaining dynamic set partitions in  $O(n)$  amortized time under all three operations by Calinescu [2].

The problem of maintaining induced set partitions can be reduced to updating an *ambiguity graph* on the  $n$  elements, where the presence of edge  $(i, j)$  indicates that elements  $i$  and  $j$  occur in different parts of at least one of the  $k$  partitions. An extensive literature exists on efficient dynamic graph algorithms [7], for such tasks as maintaining connected components under edge insertion and deletion. However, the insertion or deletion of a single  $n$ -element set partition can effect the status of  $\Theta(n^2)$  edges in such an ambiguity graph, rendering such an approach infeasible.

*Our Results.* In this paper, we present a collection of efficient and practical data structures for maintaining set partitions, as well as several generalizations of the problem. Particularly interesting is the variety of algorithmic techniques which they encompass, including classical balanced trees, randomization and random walks, suffix trees, and spanning trees of low stabbing number. In particular:

- We provide a data structure that supports the insert and report operation in optimal  $O(n)$  worst-case time, for general set partitions. Deletion takes  $O(n \lg k)$  time, where  $k$  is the number of set partitions currently in the data structure and  $n$  is the number of elements in each partition. These results are relatively straightforward, it appears nontrivial to improve all operations to  $O(n)$ , which is the best possible complexity.
- We provide randomized Monte Carlo and Las Vegas data structures that support all three operations on bipartitions in *linear* or *near-linear* expected time, although the Las Vegas bounds are amortized. The Monte Carlo data structure is *asymptotically optimal*, and the Las Vegas data structure is within a factor of  $\alpha(n)$  of optimal. We believe that our Monte Carlo data structure is particularly practical because of its simplicity. It appears widely applicable and is used as a building block for other algorithms in the paper such as the Las Vegas data structure and the geometric data structures.

- Robust classifiers compensate for noisy features by requiring more than one piece of evidence to distinguish between every pair of objects. We provide an alternate data structure that permits us to efficiently insert/delete partitions and query arbitrary pairs of elements  $\{x, y\}$  to obtain the *approximate* number of partitions currently distinguishing  $x$  from  $y$ . Insert/delete run in time  $O(n \log \log n)$  and query runs in polylogarithmic time. This data structure uses techniques from random walks on a line. Randomization and approximation appear to be powerful techniques in this setting, because, to our knowledge, the best exact deterministic techniques require  $O(n^2)$  time per insertion or deletion of partitions.
- We provide an efficient data structure for maintaining geometric set partitions, where the set partitions are induced by linear separators of points in the plane. We achieve  $O(\sqrt{n} \log n)$  time for insertion/deletion and linear report time, after an initial  $O(n^{1.5} \log n)$  preprocessing step.
- We provide the first data structures for efficiently maintaining sorted strings under character insertion/deletion. As an application, we use this structure to find the shortest run of distinguishing features from an ordering of  $k$  binary features in optimal  $O(nk)$  time.

Our paper is organized as follows. In Section 2, we present deterministic data structures for maintaining set partitions. More efficient randomized data structures are presented in Section 3. The problem of maintaining robust classifiers is discussed in Section 4. The special case of set partitions induced by geometric arrangements is discussed in Section 5. A generalization of our problem, sorting strings under character insertion/deletion is addressed in Section 6.

## 2 Basic Results: Deterministically Maintaining Set Partitions

In this section, we present an efficient data structure for deterministically maintaining set partitions under the operations *insert*, *delete*, and *report*. For *delete*, we assume that we are given a pointer to the set partition in question and hence defer the issue of retrieving these pointers to an auxiliary dictionary data structure.

*Notation.* We use the following notation throughout the paper. Let the universal set  $U = \{1, \dots, n\}$ . Each set partition  $P$  partitions  $U$  into  $\text{parts}(P)$  disjoint subsets  $P_1, \dots, P_{\text{parts}(P)}$  such that  $\cup_{i=1}^{\text{parts}(P)} P_i = U$ . Without loss of generality, we identify these subsets by the integers  $(1, \dots, \text{parts}(P))$ , respectively. Let  $\text{part}(P, i)$  denote the part of  $P$  containing element  $i$ .

**Lemma 1.** *Let  $A$  and  $B$  be set partitions of  $U = \{1, \dots, n\}$ . The induced partition (or refinement) of  $A$  and  $B$  can be computed in  $O(n)$  time.*

The proof of Lemma 1 appears in the full version of this paper. Repeated application of Lemma 1 yields a data structure that supports insertion and report in linear time but does not explicitly support deletion. A naive solution could

recompute the induced partition from scratch on each deletion by repeatedly applying Lemma 1 for a total cost of  $O(kn)$  per deletion.

**Lemma 2.** *Set partitions can be dynamically maintained such that the insertion and deletion operations take  $O(n \lg k)$  time, while report can be performed in  $O(n)$  time.*

*Proof.* We maintain a balanced binary tree whose  $k$  leaves comprise the set of input partitions  $S$ , and each intermediate node is the induced partition of its two children. Therefore, the root of this tree represents the induced partition of  $S$ , and can be produced in linear time to satisfy a report query.

Insertion and deletion can be implemented as in any balanced binary tree such as [9]. Insertion and deletion in a red-black tree require  $O(1)$  rotations in the worst case. Although each rotation affects only a constant number of nodes, the induced partitions on all  $O(\lg k)$  intermediate root-to-leaf nodes must be recomputed using Lemma 1, so that the time required for insertion and/or deletion is  $O(n \lg k)$ .  $\square$

We note that a similar structure, called a partition tree, appears in a different context in Yellin [19]. We can modify the data structure of Lemma 2 to reduce the complexity of insertion to linear:

**Theorem 1.** *Set partition can be maintained with  $O(n)$  insertion and report, and  $O(n \lg k)$  deletion.*

*Proof.* Instead of maintaining a conventionally-balanced binary tree in the structure of Lemma 2, we maintain a forest of perfectly-balanced binary trees.

As before, the leaf level of our forest contains all  $k$  of the input partitions. We number them from 1 to  $k$  according to time of insertion. On each insertion, we will add one leaf to one tree, and construct at most one additional internal node. Denoting where the leaves reside as level 0, we will add a new internal node at the  $i$ th level every  $2^i$ th insertion. For each level  $i$ , we will maintain a FIFO queue of pointers to the roots of trees of height  $i$ . In addition to this forest, we will also maintain a separate global induced partition, initially  $\{1, \dots, n\}$ .

*Inserting a Partition:* We insert the  $k$ th partition as follows:

1. Increment the partition counter  $k$ . Construct a new leaf node for partition  $P_k$ . Add the node  $k$  to the end of the level 0 queue.
2. Refine the global set partition with  $P_k$  using the algorithm of Lemma 1.
3. Define  $j$  to be the largest integer such that  $k + 1 \geq 2^j$ . Compute  $b$ , the position of the least significant 1-bit of the binary representation of  $s = k + 1 - 2^j$ . If  $s = 0$ , then  $b$  is undefined.
4. Unless  $s = 0$ , dequeue the two oldest elements  $A$  and  $B$  of level queue  $b$ . Merge the associated set partitions of  $A$  and  $B$  using the algorithm of Lemma 1. Construct a new internal node to contain the refined partition of  $A$  and  $B$ , and enqueue this node at level  $b + 1$ .

To implement report, we simply return the global result partition. To implement deletion, we replace the partition to be deleted by the partition of the last leaf to have been inserted, and delete the internal node (if any) constructed during the insertion of this leaf. We then recompute the  $O(\lg k)$  induced partitions of the internal nodes on the two effected root-to-leaf paths. Finally, we compute the new global result partition by merging all the  $O(\lg k)$  root partitions in our forest. Hence it takes  $O(n \lg k)$  time to do a deletion.

Note that the structure of forest depends only on the number of partitions currently in  $S$ , and is independent of any deletions which may have taken place. Hence to measure the complexity of insertions we can safely assume that  $k$  insertions were performed sequentially.  $\square$

### 3 Randomized Data Structures for Maintaining Set Partitions

For simplicity we assume that each partition that we insert is a *bipartition*, meaning it divides the elements into exactly two subsets. The results in this section can be generalized to the setting in which a partition breaks the elements into  $D$  sets, at an extra cost proportional to  $D$ .

We say that an event  $E$  occurs *with high probability* (*w.h.p.*) if for any  $c > 0$  there exists a proper choice of constants such that  $\Pr [E] \geq 1 - n^{-c}$ .

#### 3.1 Monte Carlo Algorithm

*Colors of Elements.* We first describe how to maintain the partition information. At each step  $t$  of the algorithm, an integer  $C_t[i]$  is associated with each element  $i$ . We call  $C_t[i]$  the *color* of element  $i$  at step  $t$ . Specifically,  $C_t[i] \in \{0, \dots, P - 1\}$ , where  $P$  has size polynomial in  $n$ . That is,  $P \in O(n^c)$ , for some constant  $c$ . We maintain the invariant that w.h.p., if two elements  $i$  and  $j$  are in the same set at step  $t$  in the (cumulative) partition  $S$  iff they have the same color, that is,  $C_t[i] = C_t[j]$ .

We also store the partitions  $P_1 \dots P_K$  that comprise  $S$ , where the partitions are ordered by increasing insertion time. We can access any element in the set of partitions in time  $O(\log K) \in O(n)$  using a balanced tree or any other basic data structure.

*Inserting a Partition.* A new partition is supplied as a 0-1 array  $A[1 \dots n]$ , where  $A[i] \in \{0, 1\}$ . We *insert* the  $k$ -th partition in step  $t$  as follows.

1.  $r_k :=$  randomly chosen integer  $\in \{1, \dots, P - 1\}$ .
2.  $P_k[i] := r_k \cdot A_k[i],$  for  $i = 1 \dots n$ .
3.  $C_t[i] := C_{t-1}[i] + P_k[i] \bmod P,$  for  $i = 1 \dots n$ .
4. Store  $P_k[1 \dots n]$  in the list of partitions.

Note that once we have calculated  $C_t[i]$ , we no longer need to store  $C_{t-1}[i]$ .

*Deleting a Partition.* We delete a partition  $P_k$  (in step  $t$ ) as follows.

1. Find  $P_k[1 \dots n]$  in the partition list.
2.  $C_t[i] := C_{t-1}[i] - P_k[i] \bmod P$ , for  $i = 1 \dots n$ .
3. Delete  $P_k[1 \dots n]$  from the partition list.

Observe that errors are one-sided. Namely, if two elements have different colors, they belong to different sets of  $S$ . An error occurs whenever two elements assigned the same color actually belong to different sets. We obtain the following theorem.

**Theorem 2.** *Insertions and deletions of partitions are executed in time  $O(n)$ . If the algorithm runs for a polynomial number of steps, then for sufficiently large  $P = O(n^c)$ , w.h.p. all insertions and deletions are executed correctly.*

The proof of Theorem 2 appears in the full version of this paper. We note that the Monte Carlo algorithm should be extremely fast because it only uses a small number of additions and subtractions. It is interesting to note that our Monte Carlo algorithm suffices for the practical application of building a tree that distinguishes all objects using a small number of probes. Our randomized scheme will never classify two objects as different which are in fact indistinguishable, and hence the only consequence of being unlucky is to add a small number of additional probes to the test set.

### 3.2 Las Vegas Algorithm

We now describe the Las Vegas algorithm for the set partition problem. The time complexity is almost the same as for the Monte Carlo algorithm except that now it is *amortized*. In order to make the algorithm Las Vegas, we remove the probability of error from the invariant that whenever two elements have the same color, they are part of the same set in  $S$ . Thus, each time we perform an insertion or deletion, we must *verify* that this invariant holds.

*Verifying an Insertion.* We first show how to verify an insertion for each step  $t$ . This entails adding an additional step at the end of the operation:

5. Verify that for all  $i, j$ ,  $(C_t[i] = C_t[j]) \implies (C_{t-1}[i] = C_{t-1}[j])$ .
6. If so, continue. If not, an *error* is found. Spawn off an independent execution or run any other alternative protocol.

Step 5 can be executed in linear time by maintaining some additional structure of the elements. Namely, in each step we will keep the elements in sorted order by increasing color. To do so, we use an additional array  $\Pi_t[1 \dots n] =$

$\pi_1^{(t)} \dots \pi_n^{(t)}$ , where  $\pi_\ell^{(t)} = j$  means that in step  $t$ ,  $j$  is the element with the  $\ell$ -th smallest color. (Ties are broken by taking into account the number of the elements.) If the ordering  $\Pi_{t-1}[1 \dots n]$  in step  $t-1$  is known, the ordering  $\Pi_t[1 \dots n]$  can be computed in linear time by merging three sorted (and interleaved) lists:

- the elements of  $\Pi_t[1 \dots n]$  whose colors do not change between step  $t-1$  and step  $t$ ,
- the elements of  $\Pi_t[1 \dots n]$  whose colors increase by  $r_k$ ,
- the elements of  $\Pi_t[1 \dots n]$  whose colors increase by  $r_k$  and then (by the rules of modular arithmetic) decrease by  $P$ .

Thus, step 5 can be broken into these substeps:

- 5a. Compute  $\pi_1^{(t)} \dots \pi_n^{(t)}$  from  $\pi_1^{(t-1)} \dots \pi_n^{(t-1)}$  by merging three lists.  
 5b. Verify that for  $\ell = 1 \dots n$ ,  $(C_t[\pi_\ell^{(t)}] = C_t[\pi_{\ell+1}^{(t)}]) \implies (C_{t-1}[\pi_\ell^{(t)}] = C_{t-1}[\pi_{\ell+1}^{(t)}])$ .

This test requires linear time because we only have to compare the color of each element  $\pi_\ell^{(t)}$  with its neighboring elements  $\pi_{\ell-1}^{(t)}$  and  $\pi_{\ell+1}^{(t)}$  in the ordering.

*Verifying a Deletion.* We now show how to verify the deletion of partition  $P_{del}[1 \dots n]$  in step  $t$ . Let  $P_{x_1} \dots P_{x_m}$  be the existing partitions in the system after  $P_{del}[1 \dots n]$  is deleted. Let  $P_{x_k}$  be the partition that appeared in  $S$  just before  $P_{del}$ .

Verifying deletions is more complicated for the following reasons. Suppose that after the deletion of a partition  $P_{del}[1 \dots n]$ , two elements  $i$  and  $j$  have the same color. We do not know *a priori* whether this is because the last partition separating  $i$  and  $j$  has been removed, or whether  $i$  and  $j$  are erroneously assigned the same color and are in fact separated by many partitions. Thus we verify deletions as follows.

4. Verify that if  $C_t[i] = C_t[j]$ , then  $i$  and  $j$  are in the same set of all partitions  $P_{x_1} \dots P_{x_n}$  (except for  $P_{del}[1 \dots n]$ ).
  5. If so, continue. If not, an *error* is found. Spawn off an independent execution or run any other alternative protocol.

We now show how to make the verification efficient. Note that step 4 could potentially be very expensive because it may involve scanning through the entire list of partitions  $P_{x_1} \dots P_{x_n}$ . Despite this, we show that the amortized cost of verifying a deletion is  $O(n\alpha(n))$ , where  $\alpha(n)$  is the inverse Ackerman function. To do this we will show that for verification, each partition is examined  $O(n)$  times and each examination requires amortized time  $O(\alpha(n))$ . As with insertions,

we maintain the elements in sorted order and this again involves merging three sorted lists. (The merge is minimally different; we now add instead of subtracting and subtract instead of adding.)

We maintain a UNION-FIND data structure for each prefix of partitions  $P_{x_1} \dots P_{x_m}$ . Thus, two elements  $i$  and  $j$  belong to the same set in the data structure UNION-FIND $^{(x_\ell)}$  if they belong to the same set in all of the partitions  $P_{x_1} \dots P_{x_\ell}$ . Because partitions are always inserted *at the end* of  $S$ , the sets in each UNION-FIND $^{(x_\ell)}$  data structure may coalesce when partitions are deleted but will never split apart. The sets can combine together at most  $n - 1$  times before a single set remains and consequently all elements have the same color. The operation FIND-SET $^{(x_\ell)}[i]$  locates the *smallest* element belonging to the same set as element  $i$  (in the UNION-FIND $^{(x_\ell)}$  data structure). The operation UNION $^{(x_\ell)}[i, j]$  *combines* the set containing  $i$  and the set containing  $j$  (in the UNION-FIND $^{(x_\ell)}$  data structure).

The algorithm appears in the full version of this paper.

**Theorem 3.** *Verifying an insertion or a deletion requires amortized time  $O(n\alpha(n))$ .*

If a verification identifies an error, then we run an alternative protocol. Because the probability of an error is polynomially small, we obtain the following theorem.

**Theorem 4.** *Insertions and deletions run in amortized time  $O(n\alpha(n))$ , both expected and w.h.p..*

## 4 Estimating the Number of Partitions Separating Elements

When building a decision tree (e.g., for OCR), one may want to maintain more detailed information besides the induced partition. In particular, for fault tolerance, one may insist that each element be separated by at least  $k$  partitions. Thus, for each pair of elements the data structure could store and return the *number* of partitions that separate the elements. Unfortunately, the naive solution to this problem requires  $O(n^2)$  for insert and delete.

On the other hand, it may not be necessary to know the *exact* number of partitions that separate elements  $i$  and  $j$ . In applications such as building decision trees, *approximate* knowledge of this number may be satisfactory. This is the problem we explore in this section.

Our data structure supports the following three operations.

- *Insert( $P, S$ )* – add a new partition  $P$  to the set of partitions  $S$ .
- *Delete( $P, S$ )* – delete existing partition  $p$  from the set of partitions  $S$ .
- *Query( $i, j, S$ )* – output an estimate of the number of partitions separating elements  $i$  and  $j$ .

We show that the operations *insert* and *delete* can be implemented to run in worst-case time  $O(n \log n)$ , or expected  $O(n)$  time. *Query* requires polylogarithmic time and returns an answer that is accurate to within any constant factor. As it is stated, this algorithm works only for bipartitions.

As in Section 3, we assign integers called *colors* to elements. However now the *distance* between two colors *approximates* the number of partitions separating elements. More specifically, each element  $i$  has  $\beta \log n$  separate colors,  $C[i][1 \dots \beta \log n]$ . Each color is modified independently.

*Inserting a partition.* A new partition is supplied as an array  $A[1 \dots n]$ , where  $A[i] \in \{0, 1\}$ . Note that each bipartition has two representations, where one representation is the complement of the other. To *insert* a partition, we independently modify each of the  $\beta \log n$  colors of the elements as follows.

- Randomly choose one of the two representations of the partitions for each of the  $\beta \log n$ , and
- add these values to the composite colors of the elements.

Thus in the  $i$ -th position some of the colors are incremented by 1 and some of the colors remain the same.

*Deleting a partition.* To delete a partition, we again modify each of the  $\beta \log n$  colors inversely to the changes on inserting the partition. For each color, we subtract the appropriate representations of the partition from the composite color so that each color either remains unmodified or decreases by 1.

*Querying elements  $i$  and  $j$ .* To *estimate* the number of colors separating elements  $i$  and  $j$ , we compare the colors of  $i$  and  $j$ . Let  $K_t$  be the number of partitions in the system at time  $t$ , and let  $E(C[i][l])$  be the expected value of  $C[i][l]$ . Notice that, for all colors of all elements, the expected value of the color is  $K_t/2$ .

However the actual colors will *deviate* from this expected value. If no sets separate  $i$  and  $j$ , then for all  $\ell = 1 \dots \beta \log n$ ,  $C[i][\ell] = C[j][\ell]$ . The less the value of  $C[i][\ell]$  and  $C[j][\ell]$  are correlated, the more sets separate  $i$  and  $j$ . Specifically, if there are  $d$  sets that separate elements  $i$  and  $j$ , then we can view the process of choosing colors for  $i$  and  $j$  as a *random walk* of length  $d$  in the following sense.

Consider the basic random walk on the integer line. A walker starts at the origin and at each step  $t$ , moves one unit to the right with probability  $1/2$  and moves one unit to the left with probability  $1/2$ . We compare this random walk with the dynamics of the algorithm. Whenever  $i$  and  $j$  are in different partitions than with probability  $1/2$ , the color of  $i$  is incremented by 1 and the color of  $j$  stays the same, and with probability  $1/2$ , the color of  $j$  is incremented by 1 and the color of  $i$  stays the same.

Thus, the probability

$$\Pr [C[i][\ell] - C[j][\ell] = z]$$

is exactly the probability that a random walk of length  $d$  ends at integer  $z$ . Thus, by examining the distribution of the colors of the elements, we can estimate the

most likely value of  $d$ . The estimation method appears in the full version of this paper. We obtain the following theorem.

**Theorem 5.** *For any error parameter  $\epsilon$  and constant  $c$ , there is a constant  $\beta$  such that the following holds with probability  $1 - 1/n^c$  (w.h.p.): If the number of sets separating elements  $i$  and  $j$  is  $d$ , then the estimate  $d'$  of  $d$  is bounded as follows:*

$$(1 - \epsilon)d \leq d' \leq (1 + \epsilon)d.$$

To implement the insert and delete operations in expected  $O(n \log \log n)$  time, we must show how to increment the  $\beta \log n$  colors by the prescribed random bits in expected  $\log \log n$  time. This can be done by maintaining these colors as  $\log k$  sets of  $2\beta$  integers, each of size  $\log n$ . The first set of integers contains the least significant bit of  $(\log n)/2$  colors, with each data bit flanked by zero bits. Adding the random bits (similarly padded) to this integer increments each of the colors simultaneously. If any carries occur, they appear as 1 bits in the padded region, and require us to increment the appropriate next significant bits. To avoid bad situations, we use nonunique representation of numbers. Each round of incrementing takes constant time, and the expected number of levels to propagate the carries is  $\log \log n$ .

## 5 Maintaining Geometric Set Partitions

Suppose the elements in each set partition were points in the plane, and that each set partition was induced by a half-plane that distinguishes between the points which lie to the left or right of the defining line. Such partitions have been previously studied. For example, Freimer, et.al [8] prove it is NP-complete to find the smallest subset of lines sufficient to completely shatter a point set, i.e. induce a complete partition of the points.

Clearly, the data structure problem can be solved by testing all of the half-planes against each point and reducing it to a non-geometric instance. However, exploiting the geometry can make the problem easier. The following operations should be supported by such a data structure.

- *Insert-line( $l, S$ )* – add a separating line  $l$  to the arrangement of partitions and points  $S$ .
- *Delete-line( $l, S$ )* – delete existing separating line  $l$  from the arrangement of partitions and points  $S$ .
- *Report( $S$ )* – report the set partition of  $U$  induced by the arrangement of partitions and points  $S$ .

A naive way to support these operations is by maintaining the arrangement of the halfplanes. Any two points in the same cell of the arrangement represent unpartitioned elements. A line insertion/deletion into the arrangement takes  $O(k + n)$  time, the former term for inserting a line in an arrangement of  $k$  lines [5] and the latter term to partition the lists of points in the split. And report takes

$O(n)$  time because we can maintain a list of non-empty cells in the arrangement with each cell maintaining a list of points in it.

This naive algorithm is faster than the algorithms in the non-geometric setting when  $k$  is  $O(n)$ , but its performance degrades for larger  $k$ . An important drawback of this algorithm is that it is not space efficient because it uses  $O(k^2)$  space to store the arrangement. We improve this naive algorithm so that insert and delete run in sublinear time. The algorithm uses spanning trees of low stabbing number and randomization.

**Theorem 6.** *Geometric set partitions can be maintained with report in  $O(n)$  and insert/delete in  $O(\sqrt{n} \log n)$  time. The data structure uses  $O(n + k)$  space and  $O(n^{1.5} \log n)$  preprocessing time. The algorithm runs correctly in polynomial time w.h.p.*

*Proof.* First we preprocess the  $n$  points to obtain a spanning tree of low stabbing number. We can find a spanning tree  $T$  of stabbing number of  $O(\sqrt{n})$  in  $O(n^{1.5} \log n)$  time [313]. We *orient* each edge of  $T$  arbitrarily. In addition we maintain an integer or *color* for each edge, initially 0.

To insert a line  $L$  we first associate an integer or *color* with  $L$ . As in the non-geometric randomized algorithms, the color is a randomly chosen integer between 1 and  $P - 1$ , where  $P$  is  $O(n^c)$  for some constant  $c$ . Then we find the set  $E$  of  $O(\sqrt{n})$  edges of tree  $T$  stabbed by  $L$ . For each edge  $e$  in  $E$  we either add or subtract the color of  $L$  modulo  $P$  from the color of  $e$ . We add if  $e$  goes from left to right of  $L$  and subtract otherwise. Finding the set  $E$  takes  $O(\sqrt{n} \log n)$  time [3], and hence insertions take  $O(\sqrt{n} \log n)$  time.

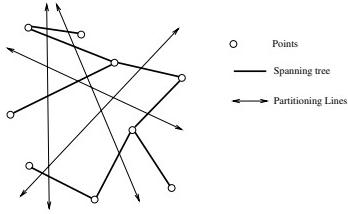
To delete a line  $L$  we find the set  $E$  of  $O(\sqrt{n})$  edges of tree  $T$  stabbed by  $L$ . Then we subtract or add modulo  $P$  the color of  $L$  from the color of each edge  $e$  of the set  $E$ . We subtract if  $e$  goes from left to right of  $L$  and add otherwise. As for insertions, finding the set  $E$  and hence deletions take  $O(\sqrt{n} \log n)$  time.

To report we start with any node of  $T$  and assign it an integer or *color* 0. We then traverse the tree and assign each node of  $T$  a color as follows. If a node has color  $c_0$  and has an outgoing edge  $e_1$  with color  $c_1$  then the target of  $e$  gets a color  $c_0 + c_1 \bmod (P)$ . Similarly for an incoming edge  $e_2$  with color  $c_2$  the source node gets a color  $c_0 - c_1 \bmod (P)$ . After assigning each node with a color, we radix sort them by color and report the partition of nodes induced by their colors. Clearly, this can be performed in  $O(n)$  time.

Note that this works because for two points in the same cell of arrangement of lines, the unique path in  $T$  between them will intersect any line an even number of times. See Figure 1. Thus while moving from one point to another we would add and subtract colors of the lines an equal number of times and hence points in the same cell will have the same color. On the other hand, if two points are in different cells the unique path in  $T$  between them would intersect at least one line odd number of times. Hence w.h.p. the two points will have different colors.

Details appear in the full paper.  $\square$

Although the off-line version of constructing the induced set partition of  $k$  set partitions of  $n$  elements can easily be solved in optimal  $\Theta(kn)$  time by repeated



**Fig. 1.** Monte Carlo algorithm for geometric set partitioning.

application of Lemma 1, the geometric version of the off-line problem is more interesting. Here we are given a set of  $k$  lines and  $n$  points, and seek to determine the induced set partition of them. We obtain the following theorems. The proofs appear in the full version.

**Theorem 7.** *The induced set partition of  $k$  lines and  $n$  points in the plane can be determined in total  $O(n^{1.5} + k\sqrt{n})$  time and  $O(n + k)$  space w.h.p.*

**Theorem 8.** *The induced set partition of  $k$  lines and  $n$  points in the plane can be determined in total  $O(k^{1.5} \log^\omega k + n\sqrt{k} \log^2 k)$  time and  $O(k \log^2 k)$  space, where  $\omega$  is a constant less than 4.3.*

## 6 Maintaining Sorted Strings under Character Insertion/Deletion

The problem of determining the partition induced by a collection of  $k$  set partitions on  $\{1, \dots, n\}$  can easily be reduced to that of sorting strings. Arbitrarily assign each of the parts of each partition  $p$  a distinct number from 0 to  $\text{parts}(p) - 1$ . Define  $S_i$  to be the string of parts associated with element  $i$ , i.e.  $S_i[j] = q$  iff element  $i$  is in part  $q$  of set partition  $j$ . Elements  $x$  and  $y$  are indistinguishable iff  $S_x = S_y$ . Hence sorting strings  $\{S_1, \dots, S_n\}$  groups the elements into blocks of equivalence classes. This gives us an alternate, more general formulation which yields our original problem as a special case – maintain the sorted order of strings where we are (1) allowed to delete the  $i$ th character from each string and (2) append an extra character to each string.

In general, we have the problem of maintaining a set of strings  $S = \{S_1, \dots, S_n\}$  under the following operations:

- *Report( $S$ )* – Return the permutation of  $S$  representing the sorted order of the strings, with runs of duplicates identified.
- *Insert( $S, i, T$ )* – Insert character  $T[j]$  after the  $i$ th position of each string  $S_j$ ,  $1 \leq j \leq n$ . This increases the length of each string in  $S$  by one character.
- *Delete( $S, i$ )* – Delete the  $i$ th character of each string  $S_j$ ,  $1 \leq j \leq n$ . This decreases the length of each string in  $S$  by one character.

A data structure to implement these operations efficiently would yield a data structure for maintaining dynamic set partitions as a special case. We propose a series of data structures based on suffix trees [10] to efficiently support a restricted set of these operations. In particular, we build our data structure around Ukkonen’s linear time suffix tree construction algorithm [18]. In Ukkonen’s algorithm, suffixes are inserted into the tree from left to right. Analogously, we can continue to append new characters onto the end of a string by simulating the insertion of another subsequent suffix.

We will augment this suffix tree to support constant-time least common ancestor queries. Cole and Hariharan [4] demonstrate how to maintain constant-time least common ancestor queries in trees supporting leaf insertion/deletion and edge-splitting updates.

**Theorem 9.** *A data structure can support the operations of head deletion, tail insertion, and sorted-order reporting in times  $O(n)$ , amortized  $O(n)$ , and  $O(n + k)$  respectively.*

The proof of Theorem 9 appears in the full version of this paper. The data structure of Theorem 9 can be used to efficiently obtain the shortest contiguous discriminating run of  $k$  preordered tests, which suggests a new heuristic approach to finding small sets of discriminating features. We repeatedly insert new features in the given order until they first suffice to completely discriminate the  $n$  strings. At this point, we repeatedly delete the prefix characters of each string, until the refined partition contains fewer than  $n$  parts. By interleaving these phases of insertion and deletion, and maintaining the boundaries of the shortest discriminating run encountered along the way:

**Corollary 1.** *The shortest contiguous discriminating run of  $k$  ordered tests can be computed in  $O(k(n + k))$  time.*

The presence of arbitrary deletions significantly complicates the problem of maintaining sorted order. However, we have reasonable results when the number of deletions is small:

**Theorem 10.** *A data structure can support the operations of arbitrary deletion, tail insertion, and sorted-order reporting in times  $O(n)$ , (amortized)  $O(n)$ , and  $O(dn \lg n)$  respectively, where  $d$  is the total number of deletions which have been performed.*

## Acknowledgment

We thank Alex Zelikovsky for introducing us to this problem.

## References

1. E. Arkin, H. Meijer, J. Mitchell, D. Rappaport, and S. Skiena. Decision trees for geometric objects. *Int. J. Computational Geometry and Applications*, 8:343–363, 1998.
2. Gruia Calinescu. A data structure for maintaining a partition. manuscript, 2000.
3. Bernard Chazelle, H. Edelsbrunner, M. Grigni, Leonidas J. Guibas, J. Hershberger, Micha Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. *Algorithmica*, 12:54–68, 1994.
4. R. Cole and R. Hariharan. Dynamic LCA queries on trees. In *Proc. Tenth ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 235–244, 1999.
5. Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.
6. P. Dietz, K. Mehlhorn, R. Raman, and C. Uhrig. Lower bounds for set intersection queries. In *Proc. Fourth ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 194–201, 1993.
7. J. Feigenbaum and S. Kannan. Dynamic graph algorithms. *Handbook of Discrete and Combinatorial Mathematics*, 1995.
8. R. Freimer, J. Mitchell, and C. Piatko. On the complexity of shattering using arrangements. In *CCCG: Canadian Conference in Computational Geometry*, 1990.
9. L. Guibas and R. Sedgewick. A bichromatic framework for balanced trees. In *Proc. 19th IEEE Symp. Foundations of Computer Science*, pages 8–21, 1978.
10. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
11. M. Habib, C. Paul, and L. Viennot. A synthesis on partition refinement: A useful routine for strings, graphs, boolean matrices and automata. In *Proc. Fifteenth STACS*, pages 25–38. Springer-Verlag LNCS, 1998.
12. J. Hopcroft. An  $n \log n$  algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The theory of machines and computations*, pages 189–196. Academic Press, New York, 1971.
13. J. Matoušek. More on cutting arrangements and spanning trees with low crossing number. Technical Report B-90-2, Fachbereich Mathematik, Freie Universität Berlin, Berlin, 1990.
14. R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM J. Computing*, 16:973–989, 1987.
15. G. Sazaklis, E. Arkin, J. Mitchell, and S. Skiena. Probe trees for touching character recognition. In *Proc. International Conference on Imaging Science, Systems and Technology, (CISST)*, pages 282–289, Las Vegas, NV, 1998.
16. G. Sazaklis, E. Arkin, J. S. B. Mitchell, and S. Skiena. Geometric decision trees for optical character recognition. In *Proc. of 13th Annual ACM Symposium on Computational Geometry*, pages 490–492, Nice, France, June 1997.
17. R. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22:215–225, 1975.
18. E. Ukkonen. Constructing suffix trees on-line in linear time. In *Intern. Federation of Information Processing (IFIP '92)*, pages 484–492, 1992.
19. D. Yellin. Representing sets with constant time equality testing. In *Proc. First ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 64–73, 1990.
20. D. Yellin. Algorithms for subset testing and finding maximal sets. In *Proc. Third ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 386–392, 1992.

# Fixed Parameter Algorithms for PLANAR DOMINATING SET and Related Problems

Jochen Alber<sup>\*1</sup>, Hans L. Bodlaender<sup>2</sup>, Henning Fernau<sup>1</sup>, and Rolf Niedermeier<sup>1</sup>

<sup>1</sup> Universität Tübingen, WSI für Informatik, Sand 13,  
72076 Tübingen, Fed. Rep. of Germany,

{alber,fernau,niedermeier}@informatik.uni-tuebingen.de

<sup>2</sup> Utrecht University, Department of Computer Science,  
Utrecht, The Netherlands,  
hansb@cs.uu.nl

**Abstract.** We present an algorithm for computing the domination number of a planar graph that uses  $O(c^{\sqrt{k}}n)$  time, where  $k$  is the domination number of the given planar input graph and  $c = 3^{6\sqrt{34}}$ . To obtain this result, we show that the treewidth of a planar graph with domination number  $k$  is  $O(\sqrt{k})$ , and that such a tree decomposition can be found in  $O(\sqrt{kn})$  time. The same technique can be used to show that the DISK DIMENSION problem (find a minimum set of faces that cover all vertices of a given plane graph) can be solved in  $O(c_1^{\sqrt{k}}n)$  time for  $c_1 = 2^{6\sqrt{34}}$ . Similar results can be obtained for some variants of DOMINATING SET, e.g., INDEPENDENT DOMINATING SET.

## 1 Introduction

A  $k$ -dominating set  $D$  of an undirected graph  $G$  is a set of  $k$  vertices of  $G$  such that each of the rest of the vertices has at least one neighbor in  $D$ . A minimal  $k$  such that the graph  $G$  has a  $k$ -dominating set is called the *domination number* of  $G$ .

The  $k$ -DOMINATING SET problem, i.e., the task to decide, given a graph  $G = (V, E)$  and a positive integer  $k$ , whether or not there exists a  $k$ -dominating set, is among the core problems in algorithms, combinatorial optimization, and computational complexity [11,16,19,24]. The problem is NP-complete, even when restricted to planar graphs with maximum vertex degree 3 and to planar graphs that are regular of degree 4 [16].

The approximability of the DOMINATING SET problem has received considerable attention [11,19]. It is not known and is not believed that DOMINATING SET for general graphs has a constant factor approximation algorithm (see Crescenzi and Kann [11] for details). However, the PLANAR DOMINATING SET problem (i.e., the dominating set problem restricted to planar graphs) possesses a polynomial

<sup>\*</sup> Work supported by the DFG-research project PEAL (Parameterized complexity and Exact ALgorithms), NI 369/1-1.

time approximation scheme [1]. That is, there is a polynomial time approximation algorithm with approximation factor  $1 + \epsilon$ , where  $\epsilon$  is a constant arbitrarily close to 0. However, the degree of the polynomial grows with  $1/\epsilon$ . Hence, applying the approximation scheme does not always lead to practical solutions and finding an “efficient” exact algorithm for PLANAR DOMINATING SET is therefore of interest.

Due to the hardness and relevance of DOMINATING SET, numerous papers have studied special cases of DOMINATING SET, e.g., connected dominating set, total dominating set, independent dominating set, dominating clique, and/or the complexity of the problem in special graph classes [6,8,10,15,17,18,22,26]. For example, a very recent result shows that there is a factor  $2 + \epsilon$  approximation algorithm for DOMINATING SET on the class of circle graphs [12].

Lately, it has become popular to cope with computational intractability in a different way besides approximation: parameterized complexity [14]. Here, the basic observation is that, for many hard problems, the seemingly inherent combinatorial explosion can be restricted to a “small part” of the input, the *parameter*. For instance, the VERTEX COVER problem can be solved by an algorithm with running time  $O(kn + 1.3^k)$  [9,23], where the parameter  $k$  is a bound on the maximum size of the vertex cover set we are looking for and  $n$  is the number of vertices in the given graph. The fundamental assumption is  $k \ll n$ . As can easily be seen, this yields an efficient, practical algorithm for small values of  $k$ . A problem is called *fixed parameter tractable* if it can be solved in time  $f(k)n^{O(1)}$  for an arbitrary function  $f$  which depends only on  $k$ . Unfortunately, according to the theory of parameterized complexity it is very unlikely that the DOMINATING SET problem is fixed parameter tractable. On the contrary, it was proven to be complete for  $W[2]$ , a “complexity class of parameterized intractability” (refer to Downey and Fellows [14] for any details). However, PLANAR  $k$ -DOMINATING SET is fixed parameter tractable. Downey and Fellows [13,14] state an  $O(11^k n)$  time bound for this problem, where  $n$  is the number of vertices.

In this paper, we present a drastic asymptotic improvement of this result. We show that PLANAR DOMINATING SET can be solved in time  $O(c^{\sqrt{k}} n)$  for some constant  $c$ . To the best of our knowledge, this is the first fixed parameter tractability result where the exponent of the exponential term is not growing linearly, but with the square root of the parameter. We show that a graph with a dominating set of size  $k$  has treewidth  $O(\sqrt{k})$ , and we use this to solve PLANAR DOMINATING SET using the corresponding tree decomposition of the graph. Unfortunately, the constant base  $c$  of the exponential term that appears in the running time of our algorithm still is quite large, namely  $c = 3^{6\sqrt{34}}$ . However, the authors are confident that a more refined analysis of the applied techniques can improve this constant considerably.

Our technique can also be used to significantly improve a known bound for the DISK DIMENSION problem [2,25]. The problem is defined as follows [2,25]: Given a plane graph  $G$ , i.e., a graph with a fixed embedding in the plane and a positive integer  $k$ , is there a set of at most  $k$  faces (disks), such that all of the graph vertices are covered? The problem is NP-complete [2]. Downey and

Fellows [14] gave an  $O(12^k n)$  algorithm for this problem. For a slightly more general version of the problem, Bienstock and Monma [2] showed that there is a time  $O(c^k n)$  algorithm, where  $c$  is an unspecified constant. In this paper, we give an algorithm that solves DISK DIMENSION in time  $O(c_1^{\sqrt{k}} n)$  for some constant  $c_1$ . We also discuss some variants of the DOMINATING SET problem.

## 2 Preliminaries

In this section, we provide necessary notions and some known results. We assume familiarity with basic graph-theoretical notation.

**Definition 1** A graph  $G$  is *outerplanar* if there is a crossing-free embedding of  $G$  in the plane such that all vertices are on the same face.

**Definition 2** A graph  $G$  is *r-outerplanar* if, for  $r = 1$ ,  $G$  is outerplanar or, for  $r > 1$ ,  $G$  has a planar embedding such that if all vertices on the exterior face (which form the *exterior layer*  $L_1$ ) are deleted, the connected components of the remaining graph are all at most  $(r - 1)$ -outerplanar.

In this way, we may speak of the layers  $L_1, \dots, L_r$  of an  $r$ -outerplanar graph. One easily makes the following central observation:

**Proposition 1.** *If a planar graph  $G = (V, E)$  has a  $k$ -dominating set, then it can be at most  $3k$ -outerplanar.*

The main tool we use in our algorithm is a suitable tree decomposition:

**Definition 3** Let  $G = (V, E)$  be a graph. A *tree decomposition* of  $G$  is a pair  $\langle \{X_i \mid i \in I\}, T \rangle$ , where each  $X_i$  is a subset of  $V$  and  $T$  is a tree with the elements of  $I$  as nodes. The following three properties should hold:

- $\bigcup_{i \in I} X_i = V$ ;
- for every edge  $\{u, v\} \in E$ , there is an  $i \in I$  such that  $\{u, v\} \subseteq X_i$ ;
- for all  $i, j, k \in I$ , if  $j$  lies on the path between  $i$  and  $k$  in  $T$ , then  $X_i \cap X_k \subseteq X_j$ .

The *width* of  $\langle \{X_i \mid i \in I\}, T \rangle$  equals  $\max\{|X_i| \mid i \in I\} - 1$ . The *treewidth* of  $G$  is the minimal  $k$  such that  $G$  has a tree decomposition of width  $k$ .

In [21, Table 2, page 550] or [5, Theorem 83], we can find:

**Proposition 2.** *An  $r$ -outerplanar graph has treewidth of at most  $3r - 1$ .*

Propositions 1 and 2 imply that a graph with domination number  $k$  has bounded treewidth, or, more precisely, its treewidth is bounded by  $9k - 1$ , but we will give a stronger bound later.

**Theorem 4.** *If a tree decomposition of width at most  $\ell$  of a graph is known, then a minimum dominating set can be determined in time  $3^\ell n$ , where  $n$  is the number of nodes of the tree decomposition.*

**Comments on the proof:** The theorem can be proved by using dynamic programming techniques. For each bag (i.e., each set  $X_i$  of the corresponding tree decomposition) one keeps a table. These tables store, for every vertex in the bag, the information of whether that vertex is assumed to belong to either the dominating set, the (known) set of dominated vertices, or the set of vertices whose status is unknown at the given point. Since  $|X_i| \leq \ell$ , the table size for each bag is bounded by  $3^\ell$ . See e.g., [427].

In this way, a straightforward solution to the PLANAR DOMINATING SET problem using tree decompositions leads to an algorithm which runs in time  $O(3^{9k}n)$ . (For a graph  $G = (V, E)$ , there always is a tree decomposition with optimal width and with at most  $|V|$  nodes.) Downey and Fellows [1314] suggested an idea that leads to a faster search tree algorithm. They state an algorithm with running time  $O(11^k n)$  (without using tree decompositions).

In what follows, we show that a graph with a  $k$ -dominating set has tree-width  $O(\sqrt{k})$ . Combining this with Theorem 4 gives a significant asymptotic improvement of the result of Downey and Fellows.

To understand the following technique, it is helpful to consider the concept of a *layer decomposition* of an  $r$ -outerplanar graph  $G$ . It is a forest of height  $r$  which is defined as follows: the nodes of the trees are sets of vertices of  $G$  and the different trees correspond to different components of  $G$ . In general, the  $i$ th layer of the layer decomposition forest defines a set of vertices  $L_i$ , namely the  $i$ th layer of  $G$ .

Consider now the  $i$ th layer of the forest, i.e., the nodes of level  $i$  in the decomposition forest, consisting of, possibly, several vertex sets  $C_{i,1}, \dots, C_{i,\ell_i}$ . In other words,  $L_i = \bigcup_{j=1}^{\ell_i} C_{i,j}$ . The vertex sets  $C_{i,1}, \dots, C_{i,\ell_i}$  correspond to the vertices of different components of the subgraph induced by  $L_i$ . We refer to  $C_{i,j}$  as a *layer-component*. In particular, the first layer consists of layer-components each of which equals the vertices from  $L_1$  of one particular component.

A layer-component  $C_{i,j}$  of layer  $L_i$  is called *non-empty* if it contains vertices from layer  $L_{i+1}$  in its interior.

**Definition 5** Let  $\emptyset \neq C \subseteq C_{i,j}$  be a subset of a non-empty layer-component  $C_{i,j}$  of layer  $i$ , where  $i \geq 2$ . Then the unique cycle  $B(C)$  in layer  $L_{i-1}$ , such that  $C$  is contained in the region enclosed by  $B(C)$  and no other vertex of layer  $L_{i-1}$  is contained in this region, is called the *boundary cycle* of  $C$ .

The existence and uniqueness of such a boundary cycle  $B(C)$  is easy to see.

### 3 Domination versus Treewidth

Our algorithm is based on Theorem 4. Therefore, in the following we show that a planar graph with domination number  $k$  has treewidth of at most  $O(f(k))$ , where  $f(k)$  is a sublinear function, which we are going to determine. Here, the main idea is to find small separators of the graph and merge the tree decompositions of the resulting subgraphs. To this end, the following observation is used.

**Proposition 3.** *If a connected graph can be decomposed into components of treewidth of at most  $t$  by means of a separator of size  $s$ , then the whole graph has treewidth of at most  $t + s$ .*

The proof is quite simple: Just merge the separator to every node in each tree decomposition of width at most  $t$  which correspond to the distinct components. Then add some arbitrary connections between the trees corresponding to the components in order to form a tree decomposition of the whole graph.

For planar graphs, there is an iterated version of this observation.

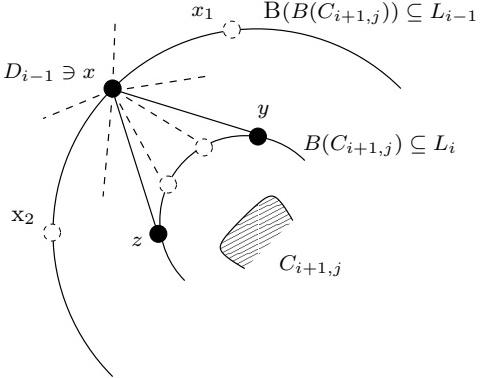
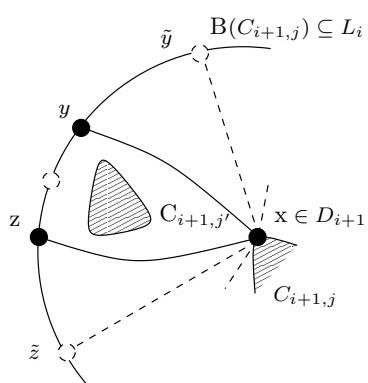
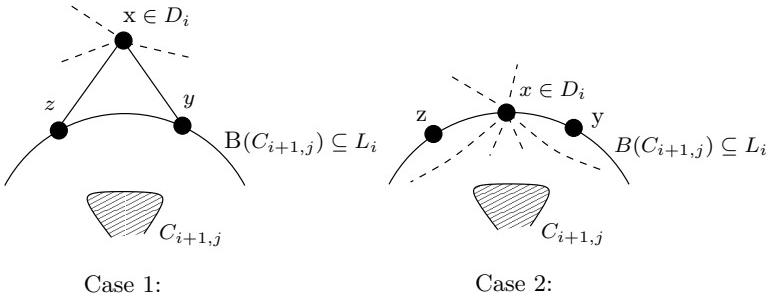
**Proposition 4.** *Let  $G$  be a planar graph with layers  $L_i$ , ( $i = 1, \dots, r$ ). For  $i = 1, \dots, \ell$ , let  $\mathcal{L}_i$  be a set of consecutive layers, i.e.  $\mathcal{L}_i = \{L_{j_i}, L_{j_i+1}, \dots, L_{j_i+n_i}\}$ , such that  $\mathcal{L}_i \cap \mathcal{L}_{i'} = \emptyset$  for all  $i \neq i'$ . Moreover, suppose  $G$  can be decomposed into components, each of treewidth of at most  $t$ , by means of separators  $S_1, \dots, S_\ell$ , where  $S_i \subseteq \bigcup_{L \in \mathcal{L}_i} L$  for all  $i = 1, \dots, \ell$ . Then  $G$  has treewidth of at most  $t + 2s$ , where  $s = \max_{i=1, \dots, \ell} |S_i|$ .*

The proof again uses the merging-techniques illustrated in the previous proposition: Suppose, w.l.o.g., the sets  $\mathcal{L}_i$  appear in successive order, i.e.  $j_i < j_{i+1}$ . For each  $i = 0, \dots, \ell$ , consider the component  $G_i$  of treewidth at most  $t$  which is cut out by the separators  $S_i$  and  $S_{i+1}$  (by default we set  $S_0 = S_{\ell+1} = \emptyset$ ). We add  $S_i$  and  $S_{i+1}$  to every node in a given tree decomposition of  $G_i$ . In order to obtain a tree decomposition of  $G$ , we successively add an arbitrary connection between the trees  $T_i$  and  $T_{i+1}$  of the so-modified tree decompositions that correspond to the subgraphs  $G_i$  and  $G_{i+1}$ .

Finally, we still have to show how to construct (in polynomial time) a tree decomposition of width  $f(k)$  matching our theoretical treewidth bound. This allows us to apply Theorem 4 to actually determine the dominating set we are aiming at.

The whole algorithm we present has time complexity  $O(3^{f(k)}n)$ . Since  $f(k) \in O(\sqrt{k})$ , this obviously gives an asymptotic improvement of the  $O(11^k n)$  algorithm presented by Downey and Fellows.

In the following, we assume that our graph has a fixed plane embedding with  $r$  layers. We show that the treewidth cannot exceed  $f(k)$  if a dominating set of size  $k$  is given.

**Fig. 1.** upper triples**Fig. 2.** lower triples**Fig. 3.** middle triples

### 3.1 Separators and Treewidth

We assume that we have a dominating set  $D$  of size at most  $k$ . Let  $t_i$  be the number of vertices of  $D_i = D \cap L_i$ . Hence,  $\sum_{i=1}^r t_i = k$ . In order to avoid case distinctions, we set  $t_0 = t_{r+1} = t_{r+2} = 0$ . Moreover, let  $c_i$  denote the number of non-empty layer-components of layer  $L_i$ .

We need some definitions for certain triples in the plane graph. These triples are defined in a way such that the union of these triples will yield separators of small size.

We define the triples for a layer  $L_i$ . The union of these triples separates vertices of layer  $L_{i-1}$  from vertices of layer  $L_{i+2}$ . For this purpose, in the following, we write  $N(x)$  to describe the set of neighbors of a vertex  $x$  and use the notion  $B(\cdot)$  for boundary cycles as introduced in Definition 5.

**Definition 6** An *upper triple* for layer  $L_i$  is associated to a non-empty layer-component  $C_{i+1,j}$  of layer  $L_{i+1}$  and a vertex  $x \in D_{i-1}$  that has a neighbor on the boundary cycle  $B(C_{i+1,j})$  (see Fig. 1). Then, clearly,  $x \in B(B(C_{i+1,j}))$ ,

by definition of a boundary cycle. Let  $x_1$  and  $x_2$  be the neighbors of  $x$  on the cycle  $B(B(C_{i+1,j}))$ . Starting from  $x_1$ , we go around  $x$  up to  $x_2$  so that we visit all neighbors of  $x$  in layer  $L_i$ . We note the neighbors of  $x$  on the boundary cycle  $B(C_{i+1,j})$ . Going around gives two outermost neighbors  $y$  and  $z$  on this boundary cycle. The triple then is the three-element set  $\{x, y, z\}$ . In case  $x$  has only a single neighbor  $y$  in  $B(C_{i+1,j})$ , the “triple” consists of only  $\{x, y\}$ .

For each non-empty layer-component  $C_{i+1,j}$  of  $L_{i+1}$  and each vertex  $x \in D_{i-1}$  with neighbors in  $B(C_{i+1,j})$ , we obtain such an upper triple.

**Definition 7** A *lower triple* for layer  $L_i$  is associated to a vertex  $x \in D_{i+1}$  and a non-empty layer-component  $C_{i+1,j'}$  of layer  $L_{i+1}$  (see Fig. 2). Suppose  $x$  lies in layer-component  $C_{i+1,j}$ . We only consider layer-components  $C_{i+1,j'}$  of layer  $L_{i+1}$  that are enclosed by the boundary cycle  $B(C_{i+1,j})$ . For each pair  $\tilde{y}, \tilde{z} \in B(C_{i+1,j}) \cap N(x)$  (where  $\tilde{y} \neq \tilde{z}$ ), we consider the path  $P_{\tilde{y}, \tilde{z}}$  from  $\tilde{y}$  to  $\tilde{z}$  along the cycle  $B(C_{i+1,j})$ , taking the direction such that the region enclosed by  $\{\tilde{z}, x\}$ ,  $\{x, \tilde{y}\}$ , and  $P_{\tilde{y}, \tilde{z}}$  contains the layer-component  $C_{i+1,j'}$ . Let  $\{y, z\} \subseteq B(C_{i+1,j}) \cap N(x)$  be the pair such that the corresponding path  $P_{y, z}$  is shortest. The triple, then, is the three-element set  $\{x, y, z\}$ . If  $x$  has no or only a single neighbor  $y$  in  $B(C_{i+1,j})$ , then the “triple” consists only of  $\{x\}$ , or  $\{x, y\}$ .

For each vertex  $x \in C_{i+1,j}$  of  $D_{i+1}$  and each non-empty layer-component  $C_{i+1,j'}$  that is enclosed by  $B(C_{i+1,j})$ , we obtain such a lower triple.

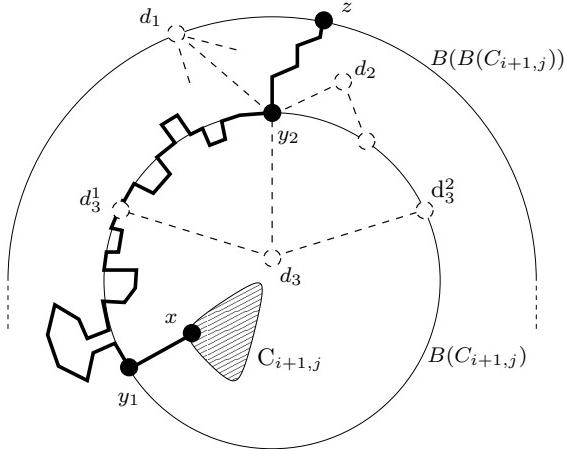
**Definition 8** A *middle triple* for layer  $L_i$  is associated to a non-empty layer-component  $C_{i+1,j}$  and a vertex  $x \in D_i$  that has a neighbor in  $B(C_{i+1,j})$  (see Fig. 3). Note that, due to the layer model, it is easy to see that a vertex  $x \in D_i$  can have at most two neighbors  $y, z$  in  $B(C_{i+1,j})$ . Depending on whether  $x$  itself lies on the cycle  $B(C_{i+1,j})$  or not, we obtain two different cases which both are illustrated in Fig. 3. In either of these cases the middle triple is defined as the set  $\{x, y, z\}$ . Again, if  $x$  has none or only a single neighbor  $y$  in  $B(C_{i+1,j})$ , then the “triple” consists only of  $\{x\}$ , or  $\{x, y\}$ .

For each non-empty layer-component  $C_{i+1,j}$  and each vertex  $x \in D_i$ , we obtain such a middle triple.

**Definition 9** We define the set  $S_i$  as the union of all upper triples, lower triples and middle triples of  $L_i$ .

In the following, we will show that  $S_i$  is a separator of the graph. Note that the upper bounds on the size of  $S_i$ , which are derived afterwards, are crucial for the upper bound on the treewidth derived later on.

**Theorem 10.** *The set  $S_i$  separates vertices of  $L_{i-1}$  and  $L_{i+2}$ .*



**Fig. 4.**  $S_i$  separates  $L_{i-1}$  and  $L_{i+2}$ .

*Proof.* Suppose there is a path  $P$  (with no repeated vertices) from layer  $L_{i+2}$  to layer  $L_{i-1}$  that avoids  $S_i$ . This clearly implies that there exists a path  $P'$  from a vertex  $x$  in a non-empty layer-component  $C_{i+1,j}$  of layer  $L_{i+1}$  to a vertex  $z \in B(B(C_{i+1,j}))$  in layer  $L_{i-1}$  which has the following two properties:

- $P' \cap S_i = \emptyset$ .
- All vertices in between  $x$  and  $z$  belong to layer  $L_i$  or to empty layer-components of layer  $L_{i+1}$ .

(This can be achieved by simply taking a suitable subpath  $P'$  of  $P$ .) Let  $y_1$  (and  $y_2$ , respectively) be the first (last) vertex along the path  $P'$  from  $x$  to  $z$  that lies on the boundary cycle  $B(C_{i+1,j}) \subseteq L_i$  (see Fig. 4).

Obviously,  $y_2$  cannot be an element of  $D$ , since, then, it would appear in a middle triple of layer  $L_i$  and, hence, in  $S_i$ . We now consider the vertex that dominates  $y_2$ . This vertex can lie in layer  $L_{i-1}$ ,  $L_i$  or  $L_{i+1}$ .

Suppose first that  $y_2$  is dominated by a vertex  $d_1 \in L_{i-1}$ . Then  $d_1$  is in  $B(B(C_{i+1,j}))$ , simply by definition of the boundary cycle (see Fig. 4). Since  $G$  is planar, this implies that  $y_2$  must be an “outermost” neighbor of  $d_1$  among all elements in  $N(d_1) \cap B(C_{i+1,j})$ . If this were not the case, then there would be an edge from  $d_1$  to a vertex on  $B(C_{i+1,j})$  that leaves the closed region bounded by  $\{d_1, y_2\}$ , the path from  $y_2$  to  $z$ , and the corresponding path from  $z$  to  $d_1$  along  $B(B(C_{i+1,j}))$ . Hence,  $y_2$  is in the upper triple of layer  $L_i$  which is associated to the layer-component  $C_{i+1,j}$  and  $d_1$ . This contradicts the fact that  $P'$  avoids  $S_i$ .

Now, suppose that  $y_2$  is dominated by a vertex  $d_2 \in D_i$  (see Fig. 4). By definition of the middle triples, this clearly implies that  $y_2$  is in the middle triple associated to  $C_{i+1,j}$  and  $d_2$ . Again, this contradicts the assumption that  $P' \cap S_i = \emptyset$ .

Consequently, the dominating vertex  $d_3$  of  $y_2$  has to lie in layer  $L_{i+1}$ . Let  $\{d_3, d_3^1, d_3^2\}$ , where  $d_3^1, d_3^2 \in N(d_3) \cap B(C_{i+1,j})$ , be the lower triple associated to layer-component  $C_{i+1,j}$  and  $d_3$  (see Fig. 4). By definition,  $C_{i+1,j}$  is contained in the region enclosed by  $\{d_3^1, d_3\}, \{d_3, d_3^2\}$  and the path from  $d_3^2$  to  $d_3^1$  along  $B(C_{i+1,j})$ , which—assuming that  $y_2 \notin \{d_3, d_3^1, d_3^2\}$ —does not hit  $y_2$  (see Fig. 4). We now observe that, whenever the path from  $y_1$  to  $y_2$  leaves the cycle  $B(C_{i+1,j})$  to its exterior, say at a vertex  $q$ , then it has to return to  $B(C_{i+1,j})$  at a vertex  $q' \in N(q) \cap B(C_{i+1,j})$ . This, however, shows that the path  $P'$  has to hit either  $d_3^1$  or  $d_3^2$  on its way from  $y_1$  to  $y_2$ . Since  $d_3^1, d_3^2 \in S_i$ , this case also contradicts the fact that  $P' \cap S_i = \emptyset$ .  $\square$

**Lemma 1.**  $|S_i| \leq 5(t_{i-1} + t_i + t_{i+1}) + 12c_{i+1}$ .

*Proof.* We give bounds for the number of vertices in upper, middle and lower triples of layer  $i$ , separately.

Firstly, we discuss the upper triples of layer  $i$ , which were associated to a non-empty layer-component  $C_{i+1,j}$  of layer  $L_{i+1}$  and a vertex  $x \in D_{i-1}$  with neighbors in  $B(C_{i+1,j})$ . Consider the bipartite graph  $G'$  which has vertices for each non-empty layer-component  $C_{i+1,j}$  and for each vertex in  $D_{i-1}$ . Whenever a vertex in  $D_{i-1}$  has a neighbor in  $B(C_{i+1,j})$ , an edge is drawn between the corresponding vertices in  $G'$ . Each edge in  $G'$ , by construction, may correspond to an upper triple of layer  $L_i$ . Note that  $G'$  is a planar bipartite graph whose bipartition subsets consist of  $t_{i-1}$  and  $c_{i+1}$  vertices, respectively. Thus, the number of edges of  $G'$  is linear in the number of vertices; more precisely, it is bounded by  $2(t_{i-1} + c_{i+1})$ . From this, we obtain an upper bound for the number of vertices in upper triples of layer  $L_i$  as follows: Potentially, each vertex of  $D_{i-1}$  appears in an upper triple and, for each edge in  $G'$ , we possibly obtain two further vertices in an upper triple. This shows that the total number of vertices in upper triples is bounded by  $t_{i-1} + 4(t_{i-1} + c_{i+1})$ .

A similar analysis can be used to show that the number of vertices in the lower triples is bounded by  $t_{i+1} + 4(t_{i+1} + c_{i+1})$  and that the number of vertices in the middle triples can be bounded by  $t_i + 4(t_i + c_{i+1})$ .

By definition of  $S_i$ , this proves our claim.  $\square$

Note that, by a more detailed investigation, the bound given in Lemma II probably can be improved. One observes, e.g., that the planar bipartite graph  $G'$ , which was constructed in the proof, has the special property that it is a “hyperplane” bipartite graph, i.e., one of the bipartition subsets can be arranged on a line and all edges of the graph lie in one halfplane of this line. This property of  $G'$  is immediate from the fact that the upper triples associated to a non-empty layer-component  $C_{i+1,j}$  lie within the boundary cycle  $B(B(C_{i+1,j}))$ . For such graphs, first investigations indicate that one can obtain better estimates on the number of their edges than the ones used in the proof of Lemma II.

A similar observation can be made for estimating the bounds for the lower triples.

**Lemma 2.**  $c_i \leq t_i + t_{i+1} + t_{i+2}$ .

*Proof.* By definition,  $c_i$  refers to only non-empty layer-components in layer  $L_i$ , i.e., there is at least one vertex of layer  $L_{i+1}$  contained within each such layer-component. Such a vertex can only be dominated by a vertex from layer  $L_i$ ,  $L_{i+1}$ , or  $L_{i+2}$ . In this way, we get the claimed upper bound.  $\square$

**Lemma 3.**  $\sum_{i=1}^r |S_i| \leq 51k$ , where  $r$  is the number of layers of the graph.

*Proof.* This follows directly when we combine the previous two lemmas.  $\square$

Consider the following three sets of vertices:  $\mathbb{S}_0 = S_1 \cup S_4 \cup S_7 \cup \dots$ ,  $\mathbb{S}_1 = S_2 \cup S_5 \cup S_8 \cup \dots$  and  $\mathbb{S}_2 = S_3 \cup S_6 \cup S_9 \cup \dots$ . As  $|\mathcal{S}_1| + |\mathcal{S}_2| + |\mathcal{S}_3| \leq 51k$ , one of these sets has size at most  $\frac{51}{3}k$ , say  $\mathcal{S}_\delta$  (with  $\delta \in \{0, 1, 2\}$ ).

**Theorem 11.** A planar graph with domination number  $k$  has treewidth of at most  $6\sqrt{34}\sqrt{k}$ .

*Proof.* Let  $\delta$  and  $\mathbb{S}_\delta$  be as obtained above. Let  $d := \frac{3}{2}\sqrt{34}$ . We now go through the sequence  $S_{1+\delta}, S_{4+\delta}, S_{7+\delta}, \dots$  and look for separators of size at most  $s(k) := d\sqrt{k}$ . Due to the estimate on the size of  $\mathbb{S}_\delta$ , such separators of size at most  $s(k)$  must appear within each  $n(k) := \frac{51}{3}d^{-1}\sqrt{k} = \frac{1}{3}\sqrt{34}\sqrt{k}$  sets in the sequence. In this manner, we obtain a set of disjoint separators of size at most  $s(k)$  each, such that any two consecutive separators from this set are at most  $3n(k)$  layers apart. Clearly, the separators chosen in this way fulfil the requirements in Proposition 4.

Observe that the components cut out in this way each have at most  $3n(k)$  layers and, hence, their treewidth is bounded by  $9n(k)$  due to Proposition 2.

Using Proposition 4, we can compute an upper bound of the treewidth  $\text{tw}$  of the originally given graph with domination number  $k$ :

$$\begin{aligned} \text{tw}(k) &\leq 2s(k) + 9n(k) \\ &= 2\left(\frac{3}{2}\sqrt{34}\sqrt{k}\right) + 9\left(\frac{1}{3}\sqrt{34}\sqrt{k}\right) \\ &= 6\sqrt{34}\sqrt{k}. \end{aligned}$$

This proves our claim.  $\square$

Observe that the tree structure of the tree decomposition obtained in the preceding proof corresponds to the structure of the layer decomposition forest.

How did we come to the constants? We simply computed the minimum of  $2s(k) + 9n(k)$  (the upper bound on the treewidth) given the bound  $s(k)n(k) \leq \frac{51}{3}k$ . This suggests  $s(k) = d\sqrt{k}$ , and  $d$  is optimal when  $2s(k) = 9n(k) = 9 \cdot \frac{51}{3}k \cdot s(k)^{-1}$ , so,  $2d = \frac{153}{d}$ , i.e.,  $d = \frac{3}{2}\sqrt{34}$ .

As already mentioned above, it seems to be possible to improve upon the bound of the treewidth by a more refined analysis.

### 3.2 Tree Decomposition

The proofs above can be turned into constructive algorithms that find tree decompositions of the stated widths. From the proof in [5] that an  $r$ -outerplanar graph has treewidth at most  $3r - 1$ , one can construct a linear time algorithm that indeed finds a tree decomposition of width  $3r - 1$  of a given  $r$ -outerplanar graph. The proofs in this paper can also be made constructive, but there is one point that needs specific attention. As we do not start with the dominating set given, we cannot construct the upper, middle, and lower triples. Instead, we compute the minimum separator between  $L_{i-1}$  and  $L_{i+2}$  directly, and use that set instead of  $S_i$  as defined in the proof of Section B.1. Such a minimum separator can be computed with well known techniques based on maximum flow (see e.g., [20]). The running time to find one such separator is  $O(sn')$ , where  $s$  is the size of the separator, and  $n'$  the number of vertices that are involved. The total time to find all separators, stopping when separators become so large that they will not be used further in the algorithm, can be bounded by  $O(\sqrt{kn})$ .

**Theorem 12.** *The PLANAR DOMINATING SET problem can be solved in time  $O(c^{\sqrt{k}}n)$ , where  $k$  is the domination number of the given graph of size  $n$ , and  $c = 3^{6\sqrt{34}}$ .*

*Proof.* A tree decomposition of width  $6\sqrt{34}\sqrt{k}$  of  $G$  can be constructed in  $O(\sqrt{kn})$  time. (If  $k$  is not known in advance, then an  $O(\sqrt{kn})$  time algorithm is still possible for this step, using detailed bookkeeping techniques. Otherwise, one can try different values of  $k$  – this can be done at the cost of an extra multiplicative factor of  $O(\log k)$  by using binary search.) Then, this tree decomposition can be used to solve the DOMINATING SET problem, as described in Theorem 4.  $\square$

The constant  $c$  above is  $3^{6\sqrt{34}}$ , which is rather large. However, a more refined analysis will help to reduce this constant significantly. Moreover, it is a worst case estimate, which might be far from what happens in practical applications.

## 4 Variations of DOMINATING SET and DISK DIMENSION

For several variations of the DOMINATING SET problem, our technique can also help to obtain algorithms with a similar running time. In particular, we have the following. Let DOMINATING SET WITH PROPERTY  $P$  be the following graph problem: Given a graph  $G = (V, E)$ , find the minimum size set  $W \subseteq V$  with  $W$  a dominating set and where property  $P(W)$  holds.

**Theorem 13.** *Suppose there is an algorithm that solves in  $O(q^\ell \cdot n)$  time the DOMINATING SET WITH PROPERTY  $P$  problem on graphs, given a tree decomposition with treewidth  $\ell$  and  $n$  nodes for some constant  $q$ . Then the DOMINATING SET WITH PROPERTY  $P$  problem can be solved in  $O(q^{d\sqrt{k}} \cdot n)$  time on planar graphs, where  $k$  is the minimum size dominating set with property  $P$  and  $d = 6\sqrt{34}$ .*

*Proof.* If the planar graph  $G$  admits a dominating set with property  $P$  of size at most  $k$ , then, clearly,  $G$  has domination number at most  $k$ . By Theorem 11, the treewidth of  $G$  is bounded by  $6\sqrt{34}\sqrt{k}$ . According to the discussion in Section 3.2, a corresponding tree decomposition can be found in time  $O(\sqrt{k}n)$ . The assumption on the existence of an  $O(q^\ell \cdot n)$  time algorithm for given tree decomposition of width  $\ell$  then yields the claim.  $\square$

Problems for which the condition of Theorem 13 holds and, hence, for which we can find such an  $O(c^{\sqrt{k}} \cdot n)$  time algorithm are, for instance, the INDEPENDENT DOMINATING SET problem, TOTAL DOMINATING SET problem, or CONNECTED DOMINATING SET problem.

We now turn our attention to the DISK DIMENSION problem (see [2, 14, 23]) which is the following: Given a plane graph  $G = (V, E)$  (i.e., a planar graph with a fixed embedding), find the minimum set of faces that cover all vertices of  $G$ . We can use the techniques established for solving DOMINATING SET WITH PROPERTY  $P$  on planar graphs to solve the DISK DIMENSION problem:

Let  $G = (V, E)$  be a plane graph. Consider the following graph: Add a vertex to each face of  $G$ , and make each such “face vertex” adjacent to all vertices that are on the boundary of that face. Let  $G' = (V', E')$  be the resulting graph. Write  $V' = V \cup V_F$ , where  $V_F$  is the set of vertices that represent a face in  $G$ .

For  $W \subseteq V'$ , we define  $P'(W) = \text{true}$  if and only if  $W \subseteq V_F$ . Then, by construction, there is a one-to-one correspondence between the sets of faces that cover the vertices of  $G$  and dominating sets in  $G'$  with property  $P'$ . In this sense, the DISK DIMENSION problem can be transformed to the DOMINATING SET WITH PROPERTY  $P'$  problem in linear time.

**Theorem 14.** *The DISK DIMENSION problem can be solved in time  $O(c_1^{\sqrt{k}} n)$ , where  $k$  is the disk dimension of the given graph of size  $n$ , and  $c_1 = 2^{6\sqrt{34}}$ .*

*Proof.* Consider the graph  $G' = (V', E')$  with  $V' = V \cup V_F$  as given above. Given a tree decomposition of width  $\ell$ , the dominating set problem with property  $P'$  can be solved in time  $O(2^\ell \cdot n)$ , similar to the dynamic programming algorithm sketched in the proof of Theorem 4. Observe that the size of the tables we have to use for each bag are smaller than for the general dominating set problem, since each vertex of  $V_F$  is either in the dominating set or not and each vertex of  $V$  is either dominated or not. This gives table size  $2^\ell$ . Theorem 13 and the one-to-one correspondence between this problem and the disk dimension problem yield the claim.  $\square$

We remark that the problem DOMINATING SET WITH PROPERTY  $P'$  as defined above is, in a bipartite variant, basically called PLANAR RED/BLUE DOMINATING SET in [14, p.38]. There, Downey and Fellows derive an  $O(12^k n)$  algorithm for this problem. In the same place, they give an  $O(12^k n)$  algorithm for DISK DIMENSION, which they call FACE COVER NUMBER FOR PLANAR GRAPHS. Hence, our observations lead to asymptotic improvements of their results.

## 5 Conclusion

In this paper, we presented a treewidth-based approach to improve the fixed parameter complexity of the PLANAR DOMINATING SET and the DISK DIMENSION problem drastically—we gained an exponential improvement over previous exact solutions for the problems. Seemingly for the first time, our results provide fixed parameter algorithms whose exponential factor has an exponent sublinear in the parameter.

In the long version of this paper, we plan to give improved estimates for the constant bases of the exponential terms. In addition, it would be interesting to investigate the practical usefulness of our result, since our estimates for the constants are worst case and very pessimistic ones. It also is interesting to see if these results can be extended to more variants of DOMINATING SET and to other graph classes (e.g., graphs of bounded genus). Another interesting open problem is how to use the techniques of this paper for the variant of the DISK DIMENSION problem, where the embedding is not given as ab input (i.e., for a given planar graph, find an embedding with minimum number of faces that cover all the vertices).

Finally, we remark that similar results on PLANAR DOMINATING SET and related problems can be obtained by making use of the small separator techniques presented in this paper together with the algorithms for outerplanarity-bounded graphs developed by Baker [1], which would also yield running times of the form  $O(c^{\sqrt{k}}n)$  for some constant  $c$ , where  $k$  is the domination number of the given graph.

## Acknowledgements

We thank Ton Kloks for many discussions on the topic of this paper.

## References

1. B. S. Baker, Approximation algorithms for NP-complete problems on planar graphs, *Journal of the ACM* **41**:153–180, 1994.
2. D. Bienstock and C. L. Monma, On the complexity of covering vertices by faces in a planar graph. *SIAM J. Comput.* **17**(1):53–76, 1988.
3. H. L. Bodlaender, A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.* **25**:1305–1317, 1996.
4. H. L. Bodlaender, Treewidth: Algorithmic techniques and results. In *Proceedings 22nd MFCS'97*, Springer-Verlag LNCS 1295, pp. 19–36, 1997.
5. H. L. Bodlaender, A partial  $k$ -arboretum of graphs with bounded treewidth. *Theor. Comp. Sci.* **209**:1–45, 1998.
6. A. Brandstädt, V. B. Le, and J. P. Spinrad, *Graph Classes: a Survey*, SIAM Monographs on Discrete Mathematics and Applications, 1999.
7. H. J. Broersma, T. Kloks, D. Kratsch, and H. Müller, Independent sets in AT-free graphs, In *Proceedings ICALP'97*, Springer-Verlag LNCS 1256, pp. 760–770, 1997.
8. M. S. Chang, Efficient algorithms for the domination problem on interval and circular arc graphs, *SIAM J. Comput.* **27**:1671–1694, 1998.

9. J. Chen, I. Kanj, and W. Jia, Vertex cover: further observations and further improvements. In *Proceedings 25th WG*, Springer-Verlag LNCS 1665, pp. 313–324, 1999.
10. D. G. Corneil and L. K. Stewart, Dominating sets in perfect graphs, *Discrete Mathematics* **86**, (1990), 145–164.
11. P. Crescenzi and V. Kann, A compendium of NP optimization problems. Available at <http://www.nada.kth.se/theory/problemList.html>, August 1998.
12. M. Damian-Iordache and S.V. Pemmaraju, A  $(2 + \epsilon)$ -approximation scheme for minimum domination on circle graphs. In *Proceedings 11th ACM-SIAM SODA 2000*, pp. 672–679.
13. R. G. Downey and M. R. Fellows, Parameterized computational feasibility. In *P. Clote, J. Remmel (eds.): Feasible Mathematics II*, pp. 219–244. Birkhäuser, 1995.
14. R. G. Downey and M. R. Fellows, *Parameterized Complexity*, Springer-Verlag, 1999.
15. R. D. Dutton and R. C. Brigham, Domination in claw-free graphs, *Congr. Num.* **132**: 69–75, 1998.
16. M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, 1979.
17. T. W. Haynes, S. T. Hedetniemi, and P. J. Slater (eds.), *Domination in graphs*, Marcel Dekker, 1998.
18. M. A. Henning, Domination in graphs, A survey, *Congr. Num.* **116**:139–179, 1996.
19. D. S. Hochbaum (ed.), *Approximation algorithms for NP-hard problems*. PWS Publishing Company, 1997.
20. A. Kanevsky, Finding all minimum-size separating vertex sets in a graph, *Networks* **23**: 533–541, 1993.
21. J. van Leeuwen, Graph Algorithms, In *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity Theory*, North Holland, 1990.
22. H. Müller and A. Brandstädt, The NP-completeness of STEINER TREE and DOMINATING SET for chordal bipartite graphs, *Theor. Comp. Sci.* **53**:257–265, 1987.
23. R. Niedermeier and P. Rossmanith. Upper bounds for Vertex Cover further improved. In *Proceedings 16th STACS*, Springer-Verlag LNCS 1563, pp. 561–570, 1999.
24. C. H. Papadimitriou, *Computational Complexity*. Addison-Wesley, 1994.
25. R. C. Read, Prospects for graph theory algorithms. *Ann. Discr. Math.* **55**:201–210, 1993.
26. J. A. Telle, Complexity of domination-type problems in graphs, *Nordic J. Comput.* **1**:157–171, 1994.
27. J. A. Telle and A. Proskurowski, Algorithms for vertex partitioning problems on partial  $k$ -trees, *SIAM J. Discr. Math.* **10**(4):529–550, 1997.

# Embeddings of $k$ -Connected Graphs of Pathwidth $k$ \*

Arvind Gupta<sup>1</sup>, Naomi Nishimura<sup>2</sup>, Andrzej Proskurowski<sup>3</sup>,  
and Prabhakar Ragde<sup>2</sup>

<sup>1</sup> School of Computing Science, Simon Fraser University, Burnaby, B.C., Canada  
[arvind@cs.sfu.ca](mailto:arvind@cs.sfu.ca)

<sup>2</sup> Department of Computer Science, University of Waterloo, Waterloo, Ontario,  
Canada [{nishi,plragde}@uwaterloo.ca](mailto:{nishi,plragde}@uwaterloo.ca)

<sup>3</sup> Department of Computer Science, University of Oregon, Eugene, Oregon, USA  
[andrzej@cs.uoregon.edu](mailto:andrzej@cs.uoregon.edu)

**Abstract.** We present  $O(n^3)$  embedding algorithms (generalizing subgraph isomorphism) for classes of graphs of bounded pathwidth, where  $n$  is the number of vertices in the graph. These include the first polynomial-time algorithm for minor containment and the first  $O(n^c)$  algorithm ( $c$  a constant independent of  $k$ ) for topological embedding of graphs from subclasses of partial  $k$ -trees. Of independent interest are structural properties of  $k$ -connected graphs of bounded pathwidth on which our algorithms are based. We also describe special cases which reduce to various generalizations of string matching, permitting more efficient solutions.

## 1 Introduction

Many fundamental problems in a diverse set of research areas can be characterized as graph embedding problems, where data is represented as graphs and patterns can be detected by finding smaller graphs in larger ones. Classic pattern-matching problems make use of the subgraph isomorphism problem, namely, the problem of determining whether there is a subgraph of an input graph  $H$  that is isomorphic to an input graph  $G$ . Viewed as an injective mapping, the subgraph isomorphism of  $G$  into  $H$  consists of a mapping of vertices of  $G$  to vertices of  $H$  so that edges of  $G$  map to corresponding edges of  $H$ . Generalizations of this mapping include topological embedding, where vertices of  $G$  map to vertices of  $H$  and edges of  $G$  map to vertex-disjoint paths in  $H$ , and minor containment, where vertices of  $G$  map to disjoint connected subgraphs of  $H$  and edges of  $G$  map to edges of  $H$ .

Subgraph isomorphism (and therefore its generalizations listed above) is known to be  $\mathcal{NP}$ -complete for general graphs, but can be solved in polynomial time for many restricted classes of graphs. Of particular interest are partial  $k$ -trees, also known as graphs of bounded treewidth (to be defined formally

---

\* Research supported by the Natural Sciences and Engineering Research Council of Canada and Communications and Information Technology Ontario.

in Section 2), algorithms for which unify many of the known polynomial-time algorithms for embedding problems. The embedding problems are also  $\mathcal{NP}$ -complete for general partial  $k$ -trees (implied by [Sys82]), even under many different connectivity and degree bounds on both graphs [GN96]. However, when both  $G$  and  $H$  are  $k$ -connected partial  $k$ -trees, there are polynomial-time algorithms for both subgraph isomorphism [MT92, DLP96] and topological embedding [GN94, GN98] but minor containment remains  $\mathcal{NP}$ -complete even for this restricted class [MT92].

The state of our knowledge about these problems is unsatisfying in a number of ways. The degree of the polynomial in the complexity of the algorithms for subgraph isomorphism and topological embedding depends on the magnitude of  $k$ , (e.g.,  $O(n^{k^2+k+5.5})$  time for topological embedding). This raises the question of whether there is an algorithm that runs in time  $O(n^c)$  for  $c$  a constant independent of  $k$ . (Such an algorithm would be unlikely if the problems were fixed-parameter intractable [DF95]). Furthermore, although polynomial-time algorithms for minor containment have been obtained when there is a degree bound [MT92, GN95], there are no previous results relating connectivity constraints and polynomial-time minor containment algorithms.

Our contributions in this paper demonstrate that for large subclasses of graphs of bounded pathwidth (a restriction on partial  $k$ -trees), there exist  $O(n^c)$  algorithms for minor containment, topological embedding, and subgraph isomorphism. The algorithms make use of a new and elegant characterization of  $k$ -connected graphs of bounded pathwidth (Section 3) which allows us to form a common framework for the algorithms (Sections 4 and 5). We show that each such graph has an essentially unique layout of the vertices on  $k$  “tracks”. These layouts, and the restrictions they imply on the structure of topological embeddings and minor containments, allow the description of intuitive algorithms with elegant proofs of correctness. In special cases (Section 6) we can exploit further structure to reduce the problems to string matching and its variants, permitting more efficient solutions. In this conference version, we omit many details; proofs of the most important and complex theorems will be briefly discussed in the text.

## 2 Preliminaries

### 2.1 Graphs, Treewidth, and Pathwidth

Throughout this paper we use standard graph-theoretic notation [BM76]. The vertex and edge sets of a graph  $G$  are denoted by  $V(G)$  and  $E(G)$  respectively; we use  $n$  to denote  $|V(G)|$ . All graphs we consider are simple and without self-loops. The set of vertices adjacent to a vertex  $v$ , the neighbourhood of  $v$ , is denoted by  $N(v)$ . A graph is  $k$ -connected if there are  $k$  vertex-disjoint paths between every pair of its vertices. Menger’s Theorem states that any separator of a  $k$ -connected graph (a set of vertices whose removal disconnects the graph) contains at least  $k$  vertices.

In this paper we deal with subclasses of graphs of bounded treewidth, as defined below.

**Definition 1.** A tree decomposition of a graph  $G$  is a pair  $(T, \chi)$  where  $T$  is a tree and  $\chi : V(T) \rightarrow 2^{V(G)}$  satisfies the following three properties: (1) for every  $a \in V(G)$ , there is an  $x \in V(T)$  such that  $a \in \chi(x)$ ; (2) for every  $e = (a, b) \in E(G)$ , there is an  $x \in V(T)$  such that  $a, b \in \chi(x)$ ; and (3) for all  $x, y, z \in V(T)$ , if  $y$  is on the path from  $x$  to  $z$  in  $T$  then  $\chi(x) \cap \chi(z) \subseteq \chi(y)$ .

The width of a tree decomposition  $(T, \chi)$  is  $\max\{|\chi(x)| - 1 : x \in V(T)\}$ . The treewidth of a graph  $G$  is the minimum width over all its tree decompositions; a graph of bounded treewidth is a graph of treewidth  $k$  for some constant  $k$ . For  $p$  a vertex of  $T$ ,  $\chi(p)$  is called a *bag* of  $T$ . A *path decomposition* of a graph is a tree decomposition in which  $T$  is a path, and the *pathwidth* of  $G$  is the minimum width over all its path decompositions. For fixed  $k$ , decompositions of treewidth or pathwidth  $k$  can be found in linear time [Bod93].

There is an equivalent definition of graphs of bounded treewidth which is often useful. A  *$k$ -tree* (sometimes *full  $k$ -tree*) is either a  $(k+1)$ -clique, or a graph formed from a smaller  $k$ -tree by adding a new vertex  $v$  of degree  $k$  adjacent to all vertices of a  $k$ -clique  $C$  ( $C$  is called the *attachment clique* of  $v$ ). A  *$k$ -leaf* is any degree- $k$  vertex of a  $k$ -tree, and a *partial  $k$ -tree* is any subgraph of a  $k$ -tree. Partial  $k$ -trees are exactly graphs of treewidth at most  $k$  [vL90].

A *full  $k$ -path* is a special type of  $k$ -tree. In its construction, we maintain the notion of a “current clique” (initially  $k$  vertices of the first  $(k+1)$ -clique, the remaining vertex being the *initial  $k$ -leaf*). When a new vertex is added (with the current clique as its attachment clique), it enters the current clique, and one vertex (possibly the new one) leaves the current clique, never to return. Note that if the new vertex immediately leaves, it is a  $k$ -leaf (the last vertex added being the *final  $k$ -leaf*). In a *proper  $k$ -path* [TUK95], the new vertex is not permitted to immediately leave (as a consequence, a proper  $k$ -path of size at least  $k+2$  has only two  $k$ -leaves).

Figure 1(a) below illustrates a full  $k$  path, where  $a$  is the initial  $k$ -leaf and  $\{b, c\}$  in the initial current clique. The vertices are added in order  $d, e, f, g, h, i$ , with  $i$  being the final  $k$ -leaf. The graph is not a proper  $k$ -path since after  $f$  is added to attachment clique  $\{d, e\}$ , it leaves immediately, allowing  $g$  to have attachment clique  $\{d, e\}$  as well.

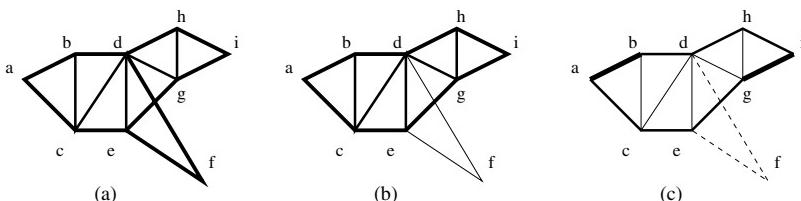


Fig. 1. Example of a full  $k$ -path

The class of *partial k-paths* (subgraphs of  $k$ -paths) is equivalent to the class of graphs of pathwidth at most  $k$  [Pro89]. The terminology we use here is in common use, though the original use of “ $k$ -path” in the previous citation was to refer to what we call a proper  $k$ -path, and other authors [KS96] have used “proper pathwidth” as a synonym for bandwidth. A *partial proper k-path* is a subgraph of a proper  $k$ -path.

We observe that a full  $k$ -path can be partitioned into the *body*, which is a proper  $k$ -path that includes the initial and final  $k$ -leaves, and *hairs*, which are the remaining  $k$ -leaves and their adjacent edges. We define an *end* of a full  $k$ -path to be the neighborhood  $h$  of a degree- $k$  vertex such that the subgraph of  $G$  induced by  $V(G) \setminus \{h\}$  has at most one component of size greater than 1. There are at most two possible ends in a full  $k$ -path of size at least  $k + 3$ , and the initial and final  $k$ -leaves each have a distinct end as their neighbor set (the *head* and *tail*, respectively). In Figure 1(b), the body is marked with thick lines and the hair with thin lines; we can view  $N(a)$  as the head and  $N(i)$  as the tail. We can view a partial  $k$ -path being decomposed in a similar fashion, where the body is a partial proper  $k$ -path; in a  $k$ -connected graph the ends will still be neighborhoods of degree- $k$  vertices.

Our algorithms will make use of a special type of path decomposition, as defined below:

**Definition 2.** *A path decomposition  $(P, \chi)$ ,  $P = p_1, \dots, p_\ell$ , of a graph  $G$  is a normalized path decomposition if (1)  $|\chi(p_i)| = k + 1$  for  $i$  odd; (2)  $|\chi(p_i)| = k$  for  $i$  even; and (3)  $\chi(p_{i-1}) \cap \chi(p_{i+1}) = \chi(p_i)$  for even  $i$ .*

Notice that  $\ell$  is always odd in this definition. It is not difficult to see that such a decomposition can be generated during the construction of a  $k$ -path; the bags of size  $k + 1$  are the attachment cliques plus respective new vertices, and the bags of size  $k$  are the current cliques. Given an already-constructed  $k$ -path, one possible construction sequence can be established by a simple linear-time scan, starting from one end.

Throughout this paper, we assume that  $G$  and  $H$  are  $k$ -connected graphs of pathwidth  $k$ , and that all path decompositions are normalized.

## 2.2 Embeddings

Each of the embeddings considered in this paper can be defined in terms of injective mappings. A *subgraph isomorphism* maps vertices of  $G$  to vertices of  $H$  and edges of  $G$  to edges of  $H$ ; it is a special case of a *topological embedding*, which maps vertices of  $G$  to vertices of  $H$  and edges of  $G$  to vertex-disjoint paths in  $H$ .

**Definition 3.** *The graph  $G$  is topologically embeddable in the graph  $H$  if there is a pair of injective functions  $f : V(G) \rightarrow V(H)$  and  $\phi : E(G) \rightarrow \{\text{paths in } H\}$  such that: (1) if  $e = (a, b) \in E(G)$  then  $\phi(e)$  has endpoints  $f(a)$  and  $f(b)$ ; and (2) for  $e, e' \in E(G)$ ,  $e \neq e'$ , the only vertices that  $\phi(e)$  and  $\phi(e')$  can have in common are their endpoints.*

A graph  $G$  is a *minor* of a graph  $H$  if a graph isomorphic to  $G$  can be formed from  $H$  by a series of edge and vertex deletions and edge contractions. Equivalently, each vertex of  $G$  is mapped to a distinct connected subgraph of  $H$  and each edge of  $G$  to a distinct edge of  $H$ , as defined below.

**Definition 4.** *The graph  $G$  is a minor of the graph  $H$  if there is a pair of functions  $(f, \xi)$  such that: (1)  $f : V(G) \rightarrow \{\text{connected subgraphs of } H\}$ ; (2)  $\xi : E(G) \rightarrow E(H)$  is injective; (3) if  $e = (a, b) \in E(G)$  then there are vertices  $u \in V(f(a))$  and  $v \in V(f(b))$  such that  $\xi(e) = (u, v)$ ; and (4) for every  $a, b \in V(G)$ ,  $a \neq b$ ,  $V(f(a)) \cap V(f(b)) = \emptyset$ . We call  $(f, \xi)$  a minor embedding of  $G$  into  $H$ .*

In discussing properties of embeddings of  $G$  into  $H$ , we will often rely on the fact that if  $G$  is embeddable in  $H$ , we can derive an *induced path decomposition* of  $G$  from the path decomposition of  $H$ ; details of this process for topological embedding were developed for  $k$ -connected partial  $k$ -trees [GN98]. To facilitate the definition, for a topological embedding  $(f, \phi)$  we define a surjective function  $\psi$  that inverts  $f$  on its image, and maps each interior vertex in  $\phi(e)$  to one of the endpoints of  $e$ . That is,  $\psi(f(a)) = a$  for every vertex  $a$  of  $G$ , and for every edge  $e = (a, b)$  of  $G$ , there is a vertex  $u$  on  $\phi(e)$  such that  $Q$  is the subpath of  $\phi(e)$  from  $f(a)$  to  $u$ , and  $R = \phi(e) \setminus Q$ , then for all vertices  $v$  of  $Q$ ,  $\psi(v) = a$  and for all vertices  $w$  of  $R$ ,  $\psi(w) = b$ .

**Definition 5.** *For  $(f, \phi)$  a topological embedding of  $G$  into  $H$ ,  $\psi$  the associated mapping of vertices in  $V(H)$  to vertices in  $V(G)$ , and  $(P, \chi)$  a path decomposition of  $H$ , we define  $\mu : V(P) \rightarrow 2^{V(G)}$  as follows:  $\mu(p) = \{\psi(u) | u \in \chi(p)\}$  and either  $f(a) = u$  for some  $a$  or  $u$  appears on  $\phi(e)$  for some  $e \in E(G)$ . We form a path  $P_G$  from  $P$  by removing each node  $p \in V(P)$  such that  $|\mu(p)| = 0$  and by replacing each subpath  $q_1, \dots, q_m$  such that  $\mu(q_i) = \mu(q_j)$  for all  $i, j$  by a single node  $q$  with  $\mu(q) = \mu(q_1)$ . We form  $\chi_G$  by restricting  $\mu$  to  $P_G$ .  $(P_G, \chi_G)$  is the path decomposition of  $G$  induced by  $(f, \phi)$ .*

A similar definition can be made for minor embedding, and we can show that for both topological embedding and minor containment,  $(P_G, \chi_G)$  is a normalized path decomposition of width  $k$ .

### 3 Track Layouts

The additional requirement of  $k$ -connectivity imposes strong restrictions on the structure of partial  $k$ -paths. We show that the body vertices can be partitioned into  $k$  *tracks* where the tracks form vertex-disjoint paths from the initial  $k$ -leaf to the final  $k$ -leaf. This partitioning is unique up to permutation of the tracks, and is independent of any specific path decomposition. Track layouts of full  $k$ -paths were considered previously [Pro84, PRS98], but our characterization of partial  $k$ -path embeddings is new.

Tracks can be extracted by examining a path decomposition of the graph. For  $(P, \chi)$  a normalized width- $k$  path decomposition of  $G$ ,  $P = p_1, \dots, p_\ell$ , we

define the *entry vertex* of  $p_i$  ( $i$  odd,  $i > 1$ ),  $\text{entry}(p_i)$ , to be the unique vertex in  $\chi(p_i) \setminus \chi(p_{i-1})$ . Similarly, the *exit vertex* of  $p_i$  ( $i$  odd,  $i < \ell$ ),  $\text{exit}(p_i)$ , is the unique vertex in  $\chi(p_i) \setminus \chi(p_{i+1})$ . A vertex  $x$  of  $G$  is a *hair vertex* (a non-initial non-final  $k$ -leaf) exactly when  $x = \text{entry}(p_i) = \text{exit}(p_i)$ , and hence when  $\text{entry}(p_i)$  or  $\text{exit}(p_i)$  is a body vertex of  $G$ ,  $\text{entry}(p_i) \neq \text{exit}(p_i)$ .

For body vertices, we can use exit and entry information to form paths in  $G$ . If  $\text{entry}(p_i)$  is not a  $k$ -leaf, it must be a neighbor of  $\text{exit}(p_i)$  (otherwise  $\chi(p_i) \setminus \{\text{entry}(p_i), \text{exit}(p_i)\}$  is a set of size  $k - 1$  separating  $\text{entry}(p_i)$  and  $\text{exit}(p_i)$  in  $G$ ). We say that  $\text{entry}(p_i)$  *replaces*  $\text{exit}(p_i)$ . A *track* is a sequence of body vertices  $a_1, a_2, \dots, a_t$  such that  $a_{i+1}$  replaces  $a_i$  for all  $1 \leq i < t$ . We use the interval notation  $[a_i, a_j]$  to represent a segment  $a_i, \dots, a_j$  of a track. The track on which a vertex  $a$  appears is denoted by  $\text{track}(a)$ . For ease of notation, we say that the  $k$ -leaves are on track 0. A *track edge* of  $G$  is any edge adjacent to the initial or final  $k$ -leaf, or any edge between vertices on the same track. Note that an internal vertex of a track is adjacent to exactly two other vertices on the same track, namely the vertex it replaces, and its own replacement. A *hair edge* is an edge adjacent to a hair. Any edge that is not a track edge or a hair edge is called a *cross edge*. Figure 1(c) illustrates track edges (thick lines), hair edges (dashed lines) and cross edges (thin lines). The lemma below is a consequence of the definition of tracks, normalized path decompositions, and  $k$ -connectivity:

**Lemma 1.** *In a path decomposition  $(P, \chi)$  of  $G$ , there exists exactly one vertex from each track in  $\chi(p_i)$  for  $i$  even.*  $\square$

We can view a layout of the vertices of  $G$  as starting with an initial  $k$ -leaf at the leftmost point, a final  $k$ -leaf at the rightmost point, and each track stretched out as a straight line from left to right. Thus, if  $b$  replaces  $a$  then we say that  $a$  is the *track predecessor* of  $b$  and that  $b$  is the *track successor* of  $a$ . The *position* of a vertex  $a$  on a track, denoted  $\text{position}(a)$ , is defined as follows: each vertex in the head is in position 1 on its track, and if  $b$  replaces  $a$ , then  $\text{position}(b) = \text{position}(a) + 1$ . Moreover, for  $\text{track}(a) = \text{track}(b)$  and  $\text{position}(a) < \text{position}(b)$ ,  $a$  is to the *left* of  $b$  and  $b$  is to the *right* of  $a$ . A *track layout* of a graph  $G$  of pathwidth  $k$  is the numbering of tracks by 1 through  $k$  and the association of each vertex  $a$  with a pair  $(\text{track}(a), \text{position}(a))$ .

Our algorithms proceed by attempting to create an embedding by mapping a track layout of  $G$  to a track layout of  $H$ . The mapping will be particularly useful if we select the track layout of  $G$  that “corresponds” to the track layout of  $H$ .

The general idea of the algorithm to find a track layout is as follows: maintain a set  $S$  (initially the head vertices, labeled 1 through  $k$ ), and repeatedly find a vertex  $a$  of  $S$  with one unlabeled neighbor  $b$ ; give  $b$  the label of  $a$  and have it replace  $a$  in  $S$ . Lemma 2 below implies that the algorithm yields the same layout independent of the order in which vertices are processed. It shows that if two vertices  $a_1$  and  $a_2$  could both be considered as  $a$ , they cannot both be replaced by the same vertex  $b_1$  (which would lead to two layouts differing in the track label of  $b_1$ ). Its proof gives an idea of the kinds of connectivity arguments important in the proofs omitted from this conference version.

**Lemma 2.** Let  $G$  be a  $k$ -connected graph of pathwidth  $k$  and  $(P', \lambda)$  be a proper prefix of at least one normalized path decomposition of  $G$ , where  $P' = (p_1, \dots, p_s)$  for  $s$  odd. Then if  $a_1 \in \lambda(p_s)$ ,  $N(a_1) \setminus \cup_{1 \leq i \leq s} \lambda(p_i) = \{b_1\}$ ,  $a_2 \in \lambda(p_s)$ ,  $N(a_2) \setminus \cup_{1 \leq i \leq s} \lambda(p_i) = \{b_2\}$ , and  $a_1 \neq a_2$ , then  $b_1 \neq b_2$ .

*Proof.* For at least one vertex  $a \in \chi(p_s)$  such a vertex  $b$  exists, namely the vertex entry  $(p_{s+1})$  for some path decomposition  $(P, \chi)$  extending  $(P', \lambda)$ . Suppose instead that for  $a_1 \neq a_2$ ,  $b_1 = b_2$ . In any normalized path decomposition  $(P, \chi)$  of which  $(P', \lambda)$  is a prefix, since  $(a_1, b_1)$  and  $(a_2, b_1)$  are both edges in  $G$ , there must exist a bag  $p_j$ ,  $j \geq s + 1$ , such that  $\{a_1, a_2, b_1\} \subseteq \chi(p_j)$  and  $\{a_1, a_2, b_1\} \cap \chi(p_{j-1}) = \{a_1, a_2\}$ . Then  $\chi(p_j) \setminus \{a_1, a_2\}$  is a set of size  $k - 1$  separating the initial and final  $k$ -leaves, violating the  $k$ -connectivity of  $G$ .  $\square$

Theorem 1 below follows as a consequence. Since either end can be the head, there are at most  $2(k!)$  different track layouts of  $G$ , each of which can be determined in linear time. Arbitrary degree- $k$  neighbors of the head and tail can be identified as the initial and final  $k$ -leaf, with all other degree- $k$  neighbors being designated as hairs.

**Theorem 1.** For  $G$  a  $k$ -connected graph of pathwidth  $k$ , for each head  $h$  of  $G$  and each numbering (permutation)  $\pi$  of the tracks, there is a unique track layout  $(h, \pi)$  of  $G$ , which can be generated in linear time.  $\square$

Although a track layout imposes a total order on the vertices of a particular track, in general it provides only a partial order on the vertices of the entire graph. Given a track decomposition starting from a particular head  $h$ ,  $a$  comes before  $b$  in the partial order if either  $a$  precedes  $b$  on the same track, or if there is an edge from  $a$  to a vertex to the left of  $b$  on the track of  $b$ . This order reflects the fact that in any path decomposition with head  $h$ ,  $a$  must appear in a bag before  $b$ . The partial order precludes the existence in the track layout of a *transposition*, namely a pair of edges  $(a_1, b_2), (b_1, a_2)$  with four distinct endpoints, where  $a_1$  (respectively  $b_1$ ) is on the same track as and to the left of  $a_2$  (respectively  $b_2$ ). This is important in proving the correctness of our algorithms.

We use  $N_t(a)$  to denote the set of neighbors of  $a$  on track  $t$ , and where appropriate we generalize the function to  $N_t(A)$  for  $A$  a set of vertices.

## 4 Topological Embedding Algorithm

Our algorithm takes an initial injective mapping  $f$  of vertices of  $G$  to vertices of  $H$  and iteratively refines the mapping until it forms an embedding of  $G$  into  $H$  or fails. Throughout the execution of the algorithm, the possible mappings considered will be constrained by track layouts of  $G$  and  $H$ . By choosing an arbitrary path decomposition of  $H$ , we can fix a total order on the vertices of  $H$  and a specific track layout. If  $G$  is embeddable in  $H$ , one of the  $2(k!)$  track layouts of  $G$  will be associated with the path decomposition of  $G$  induced by the embedding.

We initially restrict our focus to  $G$  and  $H$   $k$ -connected partial proper  $k$ -paths, and then we discuss extensions to more general situations. The track layouts of  $G$  and  $H$  can be exploited in the discussion of an embedding of  $G$  into  $H$ ; Lemma 3 below shows that track numbers and cross edges are preserved under the embeddings. A topological embedding  $(f, \phi)$  from  $G$  into  $H$  that satisfies the conditions in the lemma is said to be a *topological embedding with respect to layouts*  $(h_G, \pi_G)$  and  $(h_H, \pi_H)$ . Lemma 3 can be proved by showing that the violation of any of the conditions of the lemma allows us, by using induced path decompositions, to find a bag in the path decomposition of  $G$  violating Lemma 1.  $\square$

**Lemma 3.** *For  $G$  and  $H$   $k$ -connected partial proper  $k$ -paths,  $G$  is topologically embeddable in  $H$  if and only if there exists a topological embedding  $(f, \phi)$  and track layouts  $(h_G, \pi_G)$  of  $G$  and  $(h_H, \pi_H)$  of  $H$  such that for  $a_I$  the initial  $k$ -leaf and  $a_F$  the final  $k$ -leaf in  $G$ :*

1. *for all  $a \in V(G) \setminus \{a_I, a_F\}$ ,  $\text{track}(a) = \text{track}(f(a))$ ;*
2. *for each track edge  $(a, b)$  in  $G$ ,  $a \neq a_I$  and  $b \neq a_F$ ,  $\phi((a, b))$  consists of the path from  $f(a)$  to  $f(b)$  on the track of  $a$ ;*
3. *for each cross edge  $(a, b)$  in  $G$ ,  $\phi((a, b))$  consists of the edge  $(f(a), f(b))$ ; and*
4.  *$f(a_I) = u_I$ ,  $f(a_F) = u_F$ , for all edges  $(a_I, b)$ ,  $\phi((a_I, b))$  is a path from  $f(a_I)$  to  $f(b)$  with all interior vertices on the track of  $b$ , and for all edges  $(b, a_F)$ ,  $\phi((b, a_F))$  is a path from  $f(b)$  to  $f(a_F)$  with all interior vertices on the track of  $b$ .*

 $\square$ 

The algorithm starts by forming a single track layout and total order for  $H$  and all  $2(k!)$  track layouts of  $G$ . For each possible layout of  $G$ , initially we assign  $f(a) := u$ , where  $\text{track}(a) = \text{track}(u)$  and  $\text{position}(a) = \text{position}(u)$ . Given a mapping of vertices of  $G$  to vertices of  $H$ , extended in the obvious way to map sets of vertices, we say that  $a \in V(G)$  is *consistent* if for all tracks  $t$ ,  $f(N_t(a)) \subseteq N_t(f(a))$ . We next repeatedly check consistency of vertices in  $G$ . An inconsistency in which  $(a, b) \in E(G)$  but  $(f(a), f(b)) \notin E(H)$  is resolved by changing one or both of  $f(a)$  and  $f(b)$ . Consider the total ordering of edges of  $H$  between the track of  $f(a)$  and  $f(b)$  (induced by the total order on vertices of  $H$ , since there are no transpositions). The *leftmost consistent edge* with respect to  $a, b \in V(G)$  and mapping  $f$  is the leftmost edge  $(u, v)$  (under this total ordering of edges) such that  $\text{position}(f(a)) \leq \text{position}(u)$  and  $\text{position}(f(b)) \leq \text{position}(v)$ . To ensure  $(f(a), f(b)) \in E(H)$ ,  $f(a)$  is set to  $u$  and  $f(b)$  is set to  $v$  (which can be viewed as “sliding”  $f(a)$  or  $f(b)$  along its track), and we update function  $f$  by “sliding” vertices to the right of  $a$  and  $b$  as necessary to maintain the invariants below.

**Invariant A** For each  $a \in V(G)$ ,  $\text{track}(a) = \text{track}(f(a))$ .

**Invariant B** For vertices  $a$  and  $b$  in  $V(G)$  such that  $\text{track}(a) = \text{track}(b)$ , if  $\text{position}(a) < \text{position}(b)$ , then  $\text{position}(f(a)) < \text{position}(f(b))$ .

Lemma 4 follows from Lemma 3.

**Lemma 4.** *If there exists a mapping that satisfies invariants A and B such that, for each  $a \in V(G)$ ,  $a$  is consistent, then there is a topological embedding from  $G$  into  $H$ .*  $\square$

Given  $G$  topologically embeddable in  $H$  with respect to track layouts  $(h_G, \pi_G)$  of  $G$  and  $(h_H, \pi_H)$  of  $H$ , we can determine a partial order among topological embeddings associated with the track layouts, where  $(f_1, \phi_1)$  comes before  $(f_2, \phi_2)$  if for all  $a \in V(G)$ ,  $\text{position}(f_1(a)) \leq \text{position}(f_2(a))$ . Our algorithm finds the unique minimum embedding under this partial order, whose existence is guaranteed by the following lemma.

**Lemma 5.** *If  $G$  is topologically embeddable in  $H$  with respect to  $(h_G, \pi_G)$  and  $(h_H, \pi_H)$ , then there is a unique minimum  $f_m$  (with  $\phi$  defined as in Lemma 3) associated with  $(h_G, \pi_G)$  and  $(h_H, \pi_H)$ .*

*Proof.* Suppose instead there were incomparable minimal mappings  $(f_1, \phi_1)$  and  $(f_2, \phi_2)$ ; there must exist  $a$  and  $b$  in  $V(G)$  such that the following conditions all hold:  $\text{track}(a) \neq \text{track}(b)$ ,  $\text{position}(f_1(a)) < \text{position}(f_2(a))$  and  $\text{position}(f_2(b)) < \text{position}(f_1(b))$ . We can partition the vertices in  $V(G)$  into the following three sets:  $S_1 = \{a \in V(G) \mid \text{position}(f_1(a)) < \text{position}(f_2(a))\}$ ,  $S_2 = \{a \in V(G) \mid \text{position}(f_2(a)) < \text{position}(f_1(a))\}$ , and  $S_+ = \{a \in V(G) \mid \text{position}(f_1(a)) = \text{position}(f_2(a))\}$ .

We observe that there cannot exist an edge in  $E(G)$  between  $a \in S_1$  and  $b \in S_2$ , since the edges  $(f_1(a), f_1(b))$  and  $(f_2(a), f_2(b))$  form a transposition in  $H$ . Consequently, all edges are either between vertices in the same set, between  $S_1$  and  $S_+$ , or between  $S_2$  and  $S_+$ .

We can form  $f_3$  such that for  $a \in S_1$ ,  $f_3(a) = f_1(a)$ , for  $a \in S_2$ ,  $f_3(a) = f_2(a)$ , and for  $a \in S_+$ ,  $f_3(a) = f_1(a) = f_2(a)$ . Clearly all edges can be mapped, and hence  $f_3$  is an embedding violating the minimality of  $f_1$  and  $f_2$ , yielding a contradiction.  $\square$

Lemma 6 below demonstrates that the algorithm finds the minimum embedding. It is proved by considering the first hypothetical violation, namely a vertex  $a$  for which  $\text{position}(f(a)) > \text{position}(f_m(a))$ , and arguing that it was unnecessary to slide  $a$  past its location in the minimum embedding, since the edges of  $H$  required for consistency of  $a$  exist at that point.

**Lemma 6.** *If  $G$  is topologically embeddable in  $H$  with respect to track layouts  $(h_G, \pi_G)$  and  $(h_H, \pi_H)$ , then at any point during the execution of the algorithm above where these track layouts are chosen, and for any vertex  $a \in V(G)$ ,  $\text{position}(f(a)) \leq \text{position}(f_m(a))$ , where  $f_m$  is the unique minimal  $f$  (as defined in Lemma 5).*  $\square$

The algorithm implicit in Theorem 1 finds track layouts in linear time. In the topological embedding algorithm, each vertex slides forward at most  $n$  positions for a total of at most  $O(n^2)$  slides. Each slide is the consequence of a failed consistency check. To check all edges takes  $O(n)$  time, and so in  $O(n)$  time an inconsistent pair can be detected, if one exists. The work done in each slide is

$O(n)$ . Thus we have described an  $O(n^3)$  algorithm for topological embedding of  $k$ -connected partial proper  $k$ -paths.

**Theorem 2.** *For  $G$  and  $H$   $k$ -connected partial proper  $k$ -paths, it is possible to determine whether or not  $G$  is topologically embeddable in  $H$  in  $O(n^3)$  time.  $\square$*

When  $H$  can have hairs and thus is no longer proper, the situation is more complicated. A path of two hair edges in  $H$  can be used to embed a cross edge of  $G$  in the preimage of the attachment clique of that hair (which may no longer be a clique, since  $G$  is partial). Since there may be more than one candidate cross edge, ambiguity is introduced. We are able to resolve this ambiguity to obtain  $O(n^3)$  algorithms when one of  $G$  and  $H$  is a full  $k$ -path and the other a  $k$ -connected partial  $k$ -path.

## 5 Minor Embedding Algorithm

In the case of minor embedding, it would seem that vertices of  $G$  may now map to seemingly arbitrary connected subgraphs of  $H$ . However, as we will see, Lemma 9 gives a structural characterization which limits possible images of vertices. Throughout this section we assume that  $G$  and  $H$  are  $k$ -connected partial proper  $k$ -paths.

To facilitate the proof of Lemma 9, we first establish a few properties of minor embeddings. We focus first on the role of the initial and final  $k$ -leaves. Recall that the minor embedding function  $f$  maps vertices of  $G$  to connected subgraphs of  $H$ .

**Lemma 7.** *For any minor embedding  $(f, \xi)$  of  $G$  into  $H$ , for any  $a \in V(G)$ , if  $f(a)$  does not contain either  $u_I$  or  $u_F$ , then for any track layout  $(h_H, \pi_H)$  of  $H$  there exists a track  $t$  in  $(h_H, \pi_H)$  such that  $f(a)$  consists of an interval of vertices  $[u, v]$  on  $t$ .  $\square$*

*Proof.* We first demonstrate that  $f(a)$  cannot contain any cross edge  $(v, w)$  of  $H$ . Since  $v$  and  $w$  are neighbors but neither is the exit vertex of the other, in any path decomposition  $(P, \chi)$  of  $H$  there must exist a bag  $p_r$ ,  $r$  even, such that  $\{v, w\} \subseteq \chi(p_r)$ . If  $v$  and  $w$  are in  $f(a)$ , then in the induced path decomposition  $(P_G, \chi_G)$ ,  $|\chi_G(p_r)| < k$ , contradicting the  $k$ -connectivity of  $G$ .

Since  $f(a)$  forms a connected subgraph of  $H$  and contains no cross edge nor  $u_I$  nor  $u_F$ ,  $f(a)$  is an interval of vertices  $[u, v]$  on a single track  $t$  in  $H$ .  $\square$

**Lemma 8.** *Let  $(f, \xi)$  be a minor embedding of  $G$  into  $H$  and  $(P, \chi)$  be any path decomposition of  $H$ . Then for  $a \in \{a_I, a_F\}$  the initial or final vertex of a track layout of  $G$ , there must exist  $p_j$ ,  $j$  even, such that  $\chi(p_j)$  contains a vertex in  $f(b_i)$  for all  $1 \leq i \leq k$ , where  $b_1, \dots, b_k$  are the neighbors of  $a$ .  $\square$*

*Proof.* Since  $G$  is  $k$ -connected, each  $b_i$  has a neighbor in  $V(G) \setminus \{a, b_1, \dots, b_k\}$ . Each path decomposition of  $G$  must then contain a bag with  $\{a, b_1, \dots, b_k\}$  such

that  $b_1, \dots, b_k$  all appear in the next bag of the decomposition. If there is no bag  $p_r$ ,  $r$  even, in  $(P, \chi)$  such that that  $\chi(p_r)$  contains a vertex in  $f(b_i)$  for all  $1 \leq i \leq k$ , then the path decomposition of  $G$  induced by  $(f, \xi)$  fails to satisfy the above property, yielding a contradiction.  $\square$

With the aid of the preceding two lemmas, the proof of Lemma 9 is similar to, though more complicated than, the proof of Lemma 3 (the analogous lemma for topological embedding).

**Lemma 9.** *For  $G$  and  $H$   $k$ -connected partial proper  $k$ -paths,  $G$  is a minor of  $H$  only if there exist a minor embedding  $(f, \xi)$  and track layouts  $(h_G, \pi_G)$  and  $(h_H, \pi_H)$  such that, for  $a_I$  and  $a_F$  the initial and final  $k$ -leaves of  $(h_G, \pi_G)$  and  $u_I$  and  $u_F$  the initial and final  $k$ -leaves of  $(h_H, \pi_H)$ , the following conditions hold:*

1. *for all  $a \in V(G) \setminus \{a_I, a_F\}$ ,  $f(a)$  is an interval of vertices  $[\ell(f(a)), r(f(a))] = [u, v]$  on the track of  $a$ ;*
2. *for each track edge  $(a, b)$  in  $G$ ,  $\xi((a, b))$  consists of the edge from  $r(f(a))$  to  $\ell(f(b))$ ;*
3. *for each cross edge  $(a, b)$  in  $G$ ,  $\xi((a, b))$  consists of an edge from a vertex in  $f(a)$  to a vertex in  $f(b)$ ; and*
4.  *$f(a_I) = \{u_I\}$  and  $f(a_F) = \{u_F\}$ .*  $\square$

*Proof.* Since  $G$  is a minor of  $H$ , there must exist a minor embedding  $(g, \gamma)$  from  $G$  to  $H$ . Our proof proceeds by altering this embedding to form  $(f, \xi)$  and choosing track layouts which satisfy the conditions. We fix  $(h_H, \pi_H)$  and then create a layout for  $G$  and  $(f, \xi)$ . For  $a \in \{a_I, a_F\}$ , we let  $b_1, \dots, b_k$  be the neighbors of  $a$ . Since any bag in a path decomposition is a separator in the graph, as a consequence of Lemma 8 we can conclude that the  $f(b_j)$ 's separate  $f(a)$  from the remainder of the graph. To satisfy condition 4 it will suffice to consider the mapping of  $a$  and its neighbors to the prefix or suffix of  $(h_H, \pi_H)$  up to and including the  $f(b_j)$ 's.

By Lemmas 7, 8 and 11, each  $f(b_j)$  maps to a distinct track in  $(h_H, \pi_H)$ . By choosing  $(h_G, \pi_G)$  such that  $\text{track}(b_j) = \text{track}(f(b_j))$ , we satisfy condition 1 for each  $b_j$ .

We consider two cases for each  $a \in \{a_I, a_F\}$ , depending on whether or not  $g(a)$  contains a  $u \in \{u_I, u_F\}$ .

**Case 1:**  $g(a)$  contains  $u$

We can direct  $(h_G, \pi_G)$  so that  $u = u_I$  if and only if  $a = a_I$ . We then form  $f(a)$  by restricting  $g(a)$  to the single node  $u$ , and then set each  $f(b_j)$  to be the union of  $g(b_j)$  and each vertex on the path from  $u$  to the leftmost vertex in  $g(b_j)$  (that is, an initial segment of the vertices on track  $\text{track}(b_j)$ ).

**Case 2:**  $g(a)$  does not contain  $u_I$  or  $u_F$

We can conclude from Lemma 7 that  $g(a)$  consists of an interval  $[u, v]$  on a track  $t$  in  $H$ . By condition 1 applied to the  $b_j$ 's we can conclude that there is a path from  $g(a)$  to  $u \in \{u_I, u_F\}$  which contains no node in  $g(b_j)$  for  $1 \leq j \leq k$ . We choose  $(h_G, \pi_G)$  so that  $u = u_I$  if and only if  $a = a_I$ .

To create  $(f, \xi)$ , we set  $f(a) = u$ . We can then extend the images of the  $b_j$ 's in order to include the paths from  $u$  to each  $g(b_j)$ . In particular, for  $b_r$  such that  $\text{track}(g(b_r)) = \text{track}(g(a))$ , we set  $f(b_r)$  to be the union of  $g(b_r)$ ,  $g(a)$ , and the path from  $g(a)$  to  $u$  on  $\text{track}(g(a))$ . For each other  $b_j$  such that  $j \neq r$ , we set  $f(b_j)$  to be the union of  $g(b_j)$  and the path from  $g(b_j)$  to  $u$  on  $\text{track}(g(b_j))$ .

In both cases we have satisfied condition ④ and in addition conditions ① and ② for the two  $k$ -leaves and their neighbors. To see that condition ③ holds for the remaining vertices, we use an argument similar to that used to prove Lemma ③ in a proof by induction on track position. Namely, we show that if a track successor  $c$  of a  $b_j$  is mapped to a track other than  $\text{track}(b_j)$ , then there exists a bag in a path decomposition of  $H$  containing vertices in both  $f(b_j)$  and  $f(c)$ . The result follows from the fact that this will violate Lemma ④ in the induced path decomposition of  $G$ .

To complete the proof, we observe that condition ⑤ follows from the fact that  $(g, \gamma)$  is a minor embedding, and that condition ⑥ follows from condition ③.  $\square$

We can show that if there exists a mapping satisfying the following invariants (analogous to Invariants A and B for topological embedding), with all vertices in  $G$  consistent, then  $G$  is a minor of  $H$ . Furthermore, as for topological embeddings, we can determine a partial order among minor embeddings associated with track layouts  $(h_G, \pi_G)$  and  $(h_H, \pi_H)$ , where  $f_1$  comes before  $f_2$  if for all  $a \in G$ ,  $\text{position}(r(f_1(a))) \leq \text{position}(r(f_2(a)))$ , and show that there is a unique minimum.

**Invariant C** For each vertex  $a \in V(G)$ ,  $f(a) = [u, v]$  for some  $u$  and  $v$  on the track of  $a$  in  $H$ .

**Invariant D** For each vertex  $a \in V(G)$  with track successor  $b$ ,  $\text{position}(\ell(f(b))) = 1 + \text{position}(r(f(a)))$ .

The  $O(n^3)$  algorithm for determining (for  $k$ -connected partial proper  $k$ -paths  $G$  and  $H$ ) if  $G$  is a minor of  $H$  is a modification of the topological embedding algorithm. Initially, for  $a \in V(G)$ , the tentative minor embedding sets  $f(a)$  to be the single vertex at  $(\text{track}(a), \text{position}(a))$ . The algorithm then checks for inconsistencies and moves the right endpoints of intervals as necessary (by Invariant D, this defines left endpoints). As the new intervals may now overlap previously existing intervals, we may need to “slide” the right endpoints of those intervals as well, in a manner similar to the sliding of vertices to the right of an endpoint of an inconsistent edge in the topological embedding algorithm. The proofs of correctness and complexity, making use of Lemma ⑨, are also similar to those for topological embedding.

**Theorem 3.** *For  $G$  and  $H$   $k$ -connected partial proper  $k$ -paths, it is possible to determine whether or not  $G$  is a minor of  $H$  in  $O(n^3)$  time.*  $\square$

When trying to extend the algorithm to the case where  $H$  is no longer proper, we encounter the difficulty that a star of cross-edges (a set of cross-edges with one common endpoint) can map into a hair of  $H$ , and it is difficult to determine

which star to map. Again, if  $H$  is a full  $k$ -path and  $G$  a  $k$ -connected partial  $k$ -path, we can resolve this ambiguity, and the minor containment algorithm extends in the same fashion as the topological embedding algorithm.

## 6 Extensions and Open Problems

We can improve the complexity of our algorithms when there are additional constraints on  $G$  and  $H$ . To specify a particular full  $k$ -path, a track layout and a total order on the vertices (consistent with the partial order imposed by tracks) are sufficient. The neighbors of each vertex are completely determined by the total order. A full proper  $k$ -path  $G$  can thus be represented as a  $k$ -character string  $S_G$  derived from the track numbers of the entry vertices in a path decomposition of  $G$  [PRS98]; an extension of this notation allows us to handle full  $k$ -paths, as well, by associating with each entry node  $a$  the number  $h(a)$  of hairs sharing its attachment clique.

When  $G$  and  $H$  are both full proper  $k$ -paths, we can solve subgraph isomorphism by fixing a string representation of  $H$  and then executing string matching between  $S_H$  and each of the  $2(k!)$  possible string representations of  $G$ . When  $G$  and  $H$  are both full  $k$ -paths, we need to determine a matching such that  $a \in V(G)$  matches  $u \in V(H)$  if and only if  $\text{track}(a) = \text{track}(u)$  and  $h(a) \leq h(u)$ . This extension of string matching can be solved in time  $O(|V(H)|\sqrt{|V(G)|} \log(|V(G)|))$  [AF91], for a total complexity of  $O(n\sqrt{n} \log n)$ . It turns out that for topological embedding of full proper  $k$ -paths, a slight extension of this idea suffices to give an  $O(n\sqrt{n} \log n)$  algorithm, though we omit the details.

The most obvious open problem is to extend the algorithms to the case when both  $G$  and  $H$  are  $k$ -connected partial  $k$ -paths. It is not difficult to construct dynamic programming algorithms that solve the problems in  $O(n^{k+O(1)})$  time; a dynamic programming subproblem asks if a “prefix” of a fixed track layout of  $G$  (a subgraph closed under track predecessor, of which there are only  $O(n^k)$ ) can be mapped onto a particular track layout of  $H$ . These algorithms are essentially a simplification of the  $O(n^{k^2+k+5.5})$  algorithm for  $k$ -connected partial  $k$ -trees [GN94, GN98]. The goal, however, remains the removal of any function of  $k$  from the exponent. Beyond that, we suspect that the requirement of  $k$ -connectivity may yield more useful structural information for partial  $k$ -trees than has been discovered to date.

## Acknowledgements

We wish to thank Ming Li, Esko Ukkonen, and S. Muthukrishnan for references in the string-matching literature. We would also like to thank the referees for comments which improved the presentation of the paper.

## References

- AF91. A. Amir and M. Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 212–223, 1991.
- BM76. J. A. Bondy and U.S.R. Murty. *Graph Theory with Applications*. North-Holland, 1976.
- Bod93. H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the 25th Annual ACM Symposium on the Theory of Computing*, pages 226–234, 1993.
- DF95. R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness. I. Basic results. *SIAM Journal on Computing*, 24(4):873–921, August 1995.
- DLP96. A. Dessmark, A. Lingas, and A. Proskurowski. Faster algorithms for subgraph isomorphism of  $k$ -connected partial  $k$ -trees. In *Proceedings of the Fourth Annual European Symposium on Algorithms*, pages 501–513, 1996. To appear, Algorithmica.
- GN94. A. Gupta and N. Nishimura. Sequential and parallel algorithms for embedding problems on classes of partial  $k$ -trees. In *Proceedings of the Fourth Scandinavian Workshop on Algorithm Theory*, pages 172–182, 1994.
- GN95. A. Gupta and N. Nishimura. The parallel complexity of tree embedding problems. *Journal of Algorithms*, 18(1):176–200, 1995.
- GN96. A. Gupta and N. Nishimura. The complexity of subgraph isomorphism for classes of partial  $k$ -trees. *Theoretical Computer Science*, 164:287–298, 1996.
- GN98. A. Gupta and N. Nishimura. Topological embedding of  $k$ -connected partial  $k$ -trees. submitted, 1998.
- KS96. H. Kaplan and R. Shamir. Pathwidth, bandwidth and completion problems to proper interval graphs with small cliques. *SIAM Journal on Computing*, 25(3):540–561, 1996.
- MT92. J. Matoušek and R. Thomas. On the complexity of finding iso- and other morphisms for partial  $k$ -trees. *Discrete Mathematics*, 108:343–364, 1992.
- Pro84. A. Proskurowski. Separating subgraphs in  $k$ -trees: cables and caterpillars. *Discrete Mathematics*, 49:275–285, 1984.
- Pro89. A. Proskurowski. Maximal graphs of path-width  $k$  or searching a partial  $k$ -caterpillar. Technical Report UO-CIS-TR-89-17, University of Oregon, 1989.
- PRS98. A. Proskurowski, F. Ruskey, and M. Smith. Analysis of algorithms for listing equivalence classes of  $k$ -ary strings. *SIAM Journal of Discrete Mathematics*, 11(1):94–109, 1998.
- Sys82. M. M. Syslo. The subgraph isomorphism problem for outerplanar graphs. *Theoretical Computer Science*, 17:91–97, 1982.
- TUK95. A. Takahashi, S. Ueno, and Y. Kajitani. Mixed searching and proper-path-width. *Theoretical Computer Science*, 137(2):253–268, January 1995.
- vL90. J. van Leeuwen. *Handbook of Theoretical Computer Science A: Algorithms and Complexity Theory*, chapter Graph algorithms. North-Holland, Amsterdam, 1990.

# On Graph Powers for Leaf-Labeled Trees \*

Naomi Nishimura<sup>1</sup>, Prabhakar Ragde<sup>1</sup>, and Dimitrios M. Thilikos<sup>2</sup> \*\*

<sup>1</sup> Department of Computer Science, University of Waterloo, Waterloo, Ontario,  
Canada {nishi,plragde}@uwaterloo.ca

<sup>2</sup> Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de  
Catalunya, Barcelona, Spain sedthilk@lsi.upc.es

**Abstract.** We extend the well-studied concept of a graph power to that of a *k*-leaf power  $G$  of a tree  $T$ :  $G$  is formed by creating a node for each leaf in the tree and an edge between a pair of nodes if and only if the associated leaves are connected by a path of length at most  $k$ . By discovering hidden combinatorial structure of cliques and neighbourhoods, we have developed polynomial-time algorithms that, for  $k = 3$  and  $k = 4$ , identify whether or not a given graph  $G$  is a  $k$ -leaf power of a tree  $T$ , and if so, produce a tree  $T$  for which  $G$  is a  $k$ -leaf power. We believe that our structural results will form the basis of a solution for more general  $k$ . The general problem of inferring hidden tree structure on the basis of leaf relationships shows up in several areas of application.

## 1 Introduction

The results in this paper are derived from two abundant areas of research: graph powers and leaf-labeled trees. Both areas contain results of a purely theoretical nature as well as applications to such diverse areas as distributed computing [Lin92], computational biology, and mathematical psychology [BG91].

Trees are versatile in their ability to represent relations between data items stored in their nodes. In many instances, data items are stored in a subset of the nodes (typically leaves); the structure of internal nodes is dictated by measures of distance or similarity among leaves. For example, a Steiner tree is a tree of minimal length containing every point in a set of inputs; a more general formulation is known as an *X*-tree [BG91]. A fundamental problem in computational biology is the reconstruction of the *phylogeny*, or evolutionary history, of a set of species or genes, typically represented as a *phylogenetic tree* (the reader is referred to papers that review research in the area of evolutionary history [HKW99][HSW99][KW99]). In a phylogenetic tree, each leaf is labeled by a distinct known species; a tree is then formed by positing possible ancestors that might have led to this set of species.

---

\* Research supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Communications and Information Technology Ontario (CITO).

\*\* Research supported by the Ministry of Education and Culture of Spain, Grant number MEC-DGES SB98 0K148809.

By viewing the correlations between leaves as distances between nodes in a graph, we can frame the problem of forming a phylogenetic tree as the problem of forming a tree from a graph. One such correlation between graphs and trees, or more generally between graphs and graphs, arises in the notion of *graph powers*, where a graph  $G$  is the  $k$ *th power* of a graph  $H$  if nodes  $x$  and  $y$  are adjacent in  $G$  if and only if the length of the shortest path from  $x$  to  $y$  in  $H$  is at most  $k$ . Although in general it is NP-complete to recognize a graph power [MS94], it is possible to determine if a graph is the power of a tree in time  $O(n^3)$ , where  $n$  is the number of vertices in the input graph [CK98].

In this paper we introduce the notion of a  *$k$ -leaf power* of an unlabeled tree  $T$ , where a graph  $G$  is the  *$k$ -leaf power* of a tree  $T$  if there exists a vertex in  $V(G)$  for each leaf in  $T$  and an edge in  $E(G)$  between vertices  $u$  and  $v$  if and only if there is a path of length at most  $k$  between the leaves associated with  $u$  and  $v$  in  $T$ . The problem of recognizing  $k$ -leaf powers is inspired by the problem of forming a phylogenetic tree based on distance thresholds: given a graph  $G$  in which there is an edge for each pair of species at distance at most  $k$ , the tree  $T$  of which  $G$  is a  $k$ -leaf power is a phylogenetic tree in which the associated pair of leaves is guaranteed to be at distance at most  $k$ . A related concept is that of the *threshold graph* formed on a set of  $n$  nodes and a weighting function on edges by including only edges less than a set threshold; there exist algorithms to extract a tree from the graph by first finding connected components [BG91].

We derive polynomial-time algorithms for recognizing  $k$ -leaf powers for  $k = 3$  and  $k = 4$ . Our algorithms are based on the hidden structure of  $k$ -leaf powers, of independent combinatorial interest; as leaf labels do not play a part in our algorithms, our work is applicable to arbitrary trees. The complex characterizations of  $k$ -leaf powers are based on the structural properties of cliques and neighbourhoods. The properties are particularly tricky to derive in the presence of internal nodes that are not the neighbours of leaves: such nodes serve as invisible entities that subtly alter the structure of the relationships of neighbourhoods. The lemmas we prove may be helpful not only in generalizing our results to  $k > 4$ , but also in unrelated problems on graphs and trees.

We first establish properties of neighbourhoods in trees in Section 3. Next, we present a representation of the original graph as a *clique graph*, defined in Section 4. Section 5 contains polynomial-time algorithms which determine whether or not a graph  $G$  is a 3-leaf power or a 4-leaf power of a tree  $T$ , and if so, demonstrate one such  $T$ . Finally, directions for future research are discussed in Section 6. In this conference version, nearly all proofs, illustrations, and statements of auxiliary lemmas are omitted to save space. A few proofs have been included to give a flavour of the techniques used.

## 2 Preliminaries

### 2.1 Trees and $k$ -Leaf Powers

To help avoid confusion, we will refer to *vertices* in a tree  $T$  and *nodes* in its  $k$ -leaf power  $G$ . We classify each internal vertex as *visible* if it has a neighbouring leaf,

or *invisible* otherwise. Leaves are not defined to be either visible or invisible. A tree with no invisible vertices is an *ideal tree*.

The case of 2-leaf powers is not interesting; a 2-leaf power  $G$  is a set of disjoint cliques, each clique corresponding to the leaves of  $T$  adjacent to one internal vertex. Any tree formed by connecting the internal vertices yields the same 2-leaf power.

When  $k > 2$ , the set of leaves of  $T$  adjacent to an internal vertex also forms a clique in  $G$ , but these cliques may overlap and will not, in general, be maximal. It is the maximal cliques of  $G$  that hold the key to reconstructing  $T$ , and we must find elements in  $T$  that correspond to these cliques. Since a  $k$ -leaf power is an induced subgraph of a power of trees, and powers of trees are chordal [CK98], clearly the graphs we are trying to recognize are chordal. We can check for chordality and find all maximal cliques in linear time [Gav72, RTL76].

A few easy observations will simplify our task. Given a graph  $G$  which is a  $k$ -leaf power, we can treat each connected component separately, and connect the resulting trees by paths of length  $k$ . Consequently, we can assume (and will do so for the rest of the paper) that  $G$  is connected. Any tree  $T$  whose 3-leaf power is connected has no invisible vertices (as these would disconnect the 3-leaf power) and similarly, any tree  $T$  whose 4-leaf power is connected cannot have two adjacent invisible vertices.

The *distance*  $d(u, v)$  between two nodes  $u, v$  in a tree  $T$  is the number of edges in the unique path between them. We will find it convenient to define the distance  $d(u, e)$  between a node  $u$  and an edge  $e = (v, w)$  as  $(d(u, v) + d(u, w))/2$ . Note that, in a tree, this is always of the form  $j + \frac{1}{2}$  for an integer  $j$ ; intuitively, the extra half is the amount needed to “get to the center of the edge”. Similarly, we define the distance  $d(e_1, e_2)$  between two edges  $e_1$  and  $e_2$  in a tree as one more than the number of edges in the unique path between them. Intuitively, the addition is due to the two extra halves needed to “get to the center” of each edge. It is not hard to verify that the triangle inequality holds for this extended notion of distance.

**Definition 1.** For  $i$  even, the  $i$ -neighbourhood with center vertex  $v$  (respectively, center edge  $e$  for  $i$  odd) in a tree  $T$  is the set of all leaves of distance at most  $i/2$  from internal vertex  $v$  (respectively, edge  $e = (u, v)$ , where  $u$  and  $v$  are both internal vertices).

**Lemma 1.** In a  $2k$ -leaf (resp.  $2k + 1$ -leaf) power  $G$  of a tree  $T$ , the vertices of any maximal clique  $M$  of  $G$  form a  $2k$ -neighbourhood (resp.  $2k + 1$ -neighbourhood) in  $T$  of some internal vertex  $v$  (resp. edge  $e$ ).  $\square$

*Proof.* (even version) Let  $P$  be a longest path in  $T$  between two points of  $M$  and let the endpoints of  $P$  be  $u$  and  $w$ . Clearly,  $P$  has length at most  $2k$ . Let  $v$  be the midpoint of  $P$  (if  $P$  contains an even number of vertices, break the tie arbitrarily) and let  $N$  be the  $k$ -neighbourhood of  $v$ .

First, we show  $M \subseteq N$ . Let  $x$  be a vertex in  $M \setminus N$  (that is,  $d(v, x) > k$ ). Let  $z$  be the vertex at the point where the path from  $x$  to  $v$  meets  $P$ ;  $z$  divides

$P$  into two pieces. If both of these pieces have length less than  $k$ , then the path from  $x$  to either  $u$  or  $w$  is longer than  $P$ , contradicting the choice of  $P$ . So one piece has length at least  $k$ ; but then the distance from  $x$  to the endpoint of this piece exceeds  $2k$ , a contradiction. Thus no such vertex  $x$  exists.

Next, we show that  $N \subseteq M$ . Let  $y$  be an arbitrary vertex in  $N$  and  $x$  an arbitrary vertex of  $M$ . Since  $M \subseteq N$ ,  $d(x, v) \leq k$ . Thus  $d(x, y) \leq d(x, v) + d(v, y) \leq 2k$ , and  $x$  and  $y$  are connected in  $G$ . Since  $x$  was arbitrary,  $y$  is connected to every vertex in  $M$ , and by maximality of  $M$ ,  $y \in M$ .  $\square$

The converse of Lemma II fails to hold. For example, we can construct a path  $u, v, w$ , and  $x$  of internal vertices such that both  $v$  and  $x$  are invisible, and all other neighbours of  $w$  are leaves. Then, the 4-neighbourhood with center  $w$  (the leaf neighbours of  $w$ ) is a proper subset of the 4-neighbourhood with center  $v$  (the leaf neighbours of  $u$  and  $w$ ). Even in an ideal tree, vertices “close to the edge” can have nonmaximal 4-neighbourhoods, in a way quantified in the next section, where we look at the structure underlying neighbourhoods and maximal cliques.

### 3 Properties of Neighbourhoods

We will discover the hidden structure of the underlying tree of a  $k$ -leaf power by intersecting maximal cliques, which are neighbourhoods. The following technical lemma aids in characterizing the structure of intersections of neighbourhoods.

**Lemma 2.** *For  $v_1$  and  $v_2$  internal vertices or edges in a tree  $T$  such that  $d(v_1, v_2) = r$ , and  $S_i$  the  $k_i$ -neighbourhood of  $v_i$  for  $k_i \geq 2$ ,  $i \in \{1, 2\}$ , the following conditions hold:*

- (a) if  $k_1 + k_2 - 4 < 2r$ , then  $S_1 \cap S_2 = \emptyset$ ;
- (b) if  $2r \leq k_1 - k_2$ , then  $S_2 \subseteq S_1$ ;
- (c) if  $2r \leq k_2 - k_1$ , then  $S_1 \subseteq S_2$ ; and
- (d) if  $k_1 + k_2 - 4 \geq 2r > |k_1 - k_2|$ , then  $S_1 \cap S_2$  is the  $(\frac{k_1+k_2}{2} - r)$ -neighbourhood of the unique vertex/edge whose distance from  $v_1$  is  $\frac{k_1-k_2}{4} + \frac{r}{2}$  and whose distance from  $v_2$  is  $\frac{k_2-k_1}{4} + \frac{r}{2}$ .  $\square$

*Proof.* We will examine the case where  $k_1, k_2$ , and  $r$  are all even. The analysis for the other cases is very similar. Suppose that  $S_1 \cap S_2 \neq \emptyset$  and  $w$  is an arbitrary leaf in  $S_1 \cap S_2$ . Let  $v$  be the unique vertex of  $T$  that is adjacent to  $w$ . Clearly,  $d(v, v_i) \leq k_i/2 - 1$ , for  $i = 1, 2$ , and hence (a) follows from the fact that  $r = \text{dist}(v_1, v_2) \leq \frac{k_1+k_2}{2} - 2$ .

Suppose now that  $k_1 \geq 2r + k_2$  and let  $x \in S_2$ . This means that  $d(v_2, x) \leq k_2/2$ . As  $d(v_1, x) \leq d(v_1, v_2) + d(v_2, x)$ , we can conclude that  $d(v_1, x) \leq r + k_2/2 \leq k_1/2$  and  $x \in S_1$ . Therefore,  $S_2 \subseteq S_1$  and (b) follows. The proof of (c) is very similar.

We now let  $u$  be the unique vertex of  $T$  that is at distance  $\frac{k_1-k_2}{4} + \frac{r}{2}$  from  $v_1$  and distance  $\frac{k_2-k_1}{4} + \frac{r}{2}$  from  $v_2$ , and  $S$  be the  $(\frac{k_1+k_2}{2} - r)$ -neighbourhood

of  $u$ . We must show that  $S = S_1 \cap S_2$ . For  $x \in S$ ,  $d(x, u) \leq \frac{k_1+k_2}{4} - \frac{r}{2}$ . As  $d(u, v_1) = \frac{k_1-k_2}{4} + \frac{r}{2}$ , we conclude that  $d(x, v_1) \leq d(x, u) + d(u, v_1) = \frac{k_1+k_2}{4} - \frac{r}{2} + \frac{k_1-k_2}{4} + \frac{r}{2} = k_1/2$ , and  $x \in S_1$ . Similarly, we can show that  $x \in S_2$  and thus  $S \subseteq S_1 \cap S_2$ .

For  $x \notin S$ ,  $\text{dist}(x, u) > \frac{k_1+k_2}{4} - \frac{r}{2}$ . Deleting  $u$  divides  $T$  into connected components; notice that one of the  $v_i$ 's, say  $v_1$ , is not in the connected component that contains  $x$ . As  $v_1$  and  $x$  are in different connected components, clearly  $d(x, v_1) = \text{dist}(x, u) + d(u, v_1) > \frac{k_1+k_2}{4} - \frac{r}{2} + \frac{k_1-k_2}{4} + \frac{r}{2} = k_1/2$  and  $x$  cannot be in the  $k_1$ -neighbourhood of  $v_1$ . This implies that  $x \notin S_1 \cap S_2$ , and we conclude that  $S = S_1 \cap S_2$ .  $\square$

Using Lemma 2 we can easily prove the following results concerning the structure of neighbourhoods by simply setting the parameters  $k_1$ ,  $k_2$ , and  $r$ . Although stated in a general form, in this paper we apply these primarily in the case  $j = 4$ .

**Lemma 3.** *The following conditions hold for any tree  $T$  and  $j \geq 4$ :*

1. *The intersection of two distinct  $j$ -neighbourhoods is either empty or a  $j'$ -neighbourhood for  $2 \leq j' \leq j - 1$ .*
2. *No  $(j - 1)$ -neighbourhood is a subset of more than two distinct  $j$ -neighbourhoods for  $j$  even.*
3. *If a  $(j - 2)$ -neighbourhood is a subset of a  $j$ -neighbourhood, then their centers are either identical or adjacent.*
4. *The  $(j - 1)$ -neighbourhood of an edge is the intersection (union) of the  $j$ -neighbourhoods ( $(j - 2)$ -neighbourhoods) of its endpoints.*
5. *The  $(j - 2)$ -neighbourhood of a vertex of degree at least two is the intersection of the  $j$ -neighbourhoods of any two of its neighbours for  $j$  even.*
6. *Let  $S$  be the intersection of two  $3$ -neighbourhoods of two edges  $e_1, e_2$ . If  $e_1$  and  $e_2$  are adjacent then  $S$  is the  $2$ -neighbourhood of their common endpoint; otherwise  $S$  is empty.*  $\square$

We make use of terminology that distinguishes between types of vertices in a tree. For  $T'$  the tree obtained from  $T$  after two successive leaf prunings, we partition the internal vertices of  $T$  into those which are not in  $T'$  (*marginal vertices*), those which are leaves in  $T'$  (*peripheral vertices*) and those which are internal nodes in  $T'$  (*central vertices*). Any edge incident on a central vertex is a *central edge*. An edge in  $T$  is *pendant* if one of its endpoints is a leaf.

## 4 Clique Graphs and Their Properties

Our algorithms rely on the representation of graphs as directed acyclic graphs of maximal cliques and their intersections. In this section we introduce the notion of a *clique graph* and establish properties that prove useful in the development of our algorithms.

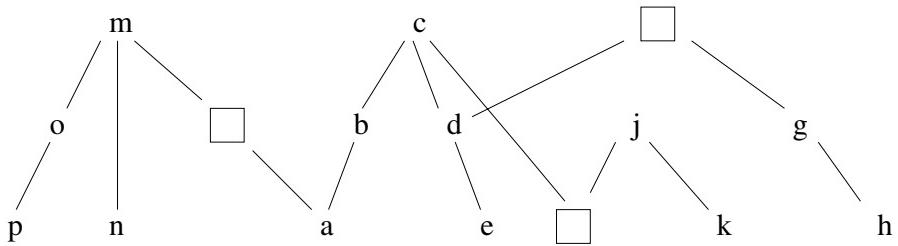
## 4.1 Clique Graphs

The *clique graph*  $C_G$  of a chordal graph  $G$  is a directed acyclic graph, whose nodes are labeled by cliques of vertices in  $G$ . The definition of  $C_G$  is given algorithmically. We build  $C_G$  by first computing a set of node labels, and then creating edges. The node label set is initialized to all maximal cliques. Then all intersections of pairs of existing node labels are added to the set. This intersection step is repeated one more time, which completes the set of node labels. A node is created for each label, and an edge added from a node  $a$  to a node  $b$  if the label of  $a$  is a subset of the label of  $b$ . In Lemma 8 we prove linear bounds on the numbers of nodes and edges of clique graphs of leaf powers, and the algorithm halts if these bounds are exceeded. This is necessary because the clique graph of an arbitrary chordal graph could have exponential size (consider a graph on vertices  $u_1, \dots, u_{n/2}, v_1, \dots, v_{n/2}$  in which there is an edge  $(u_i, u_j)$  and  $(u_i, v_j)$  for all  $i \neq j$ ). Finally, we construct the transitive reduction of the graph in a naive fashion, by checking triples of nodes  $(a, b, c)$ , and removing the edge  $(a, c)$  if edges  $(a, b)$  and  $(b, c)$  exist (this is unambiguous since our graph is a directed acyclic graph). Beyond ensuring a polynomial running time, we have not attempted to optimize this construction; further investigation of the properties of chordal graphs may improve the lemma below.

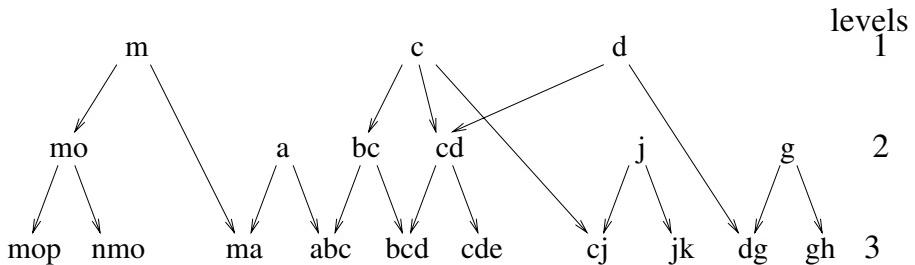
**Lemma 4.** *Given a chordal graph  $G = (V, E)$ ,  $C_G$  can be computed in time  $O(|V|^3)$ .*

*Proof.* (sketch) We first find all  $O(|V|)$  maximal cliques in time  $O(|V|^2)$  (the number of cliques and running time are a consequence of the linear-time recognition of chordal graphs by means of a perfect elimination ordering [RTL76, TY84]). Next, we form sets of intersections of sets in  $O(|V|^3)$  time. Forming the graph and its transitive reduction can be accomplished naively in cubic time.  $\square$

The label of a node  $c$  in a clique graph is its *clique graph label*, denoted  $\text{cglabel}(c)$ . We use well-known tree/DAG terminology such as *parent*, *child*, *grandparent*, *grandchild*, *descendant*, and *ancestor* to describe relationships between nodes of a clique graph. A node of the clique graph is a *border node* if it has a unique parent. Sinks (labeled by maximum cliques) are at *level*  $k - 1$  (where  $k - 2$  is the length of the longest directed path in  $C_G$ ) and any other node is at a level one less than the minimum level of its children. If a node is at level  $j$  we call it a *level- $j$  node*. We use  $C_{j,j+1}$  to denote the underlying undirected subgraph of  $C_G$  induced on edges between nodes at levels  $j$  and  $j + 1$ . Levels of nodes in a clique graph can easily be found by depth-first search, and we prove in subsequent sections that the clique graph of a  $k$ -leaf power has at most  $k - 1$  levels (for  $k = 3, 4$ ). The internal vertices of a sample input  $T$  are illustrated in Figure 1; for convenience, sets of leaf neighbours, omitted from the figure, are indicated by letter labels and invisible vertices are indicated by squares. Figure 2 depicts the clique graph  $G$  (for  $k = 4$ ) generated from the tree  $T$ . In this example, the nodes with clique graph labels  $mop$ ,  $nmo$ ,  $cde$ ,  $jk$ , and  $gh$  are all border nodes.



**Fig. 1.** Internal vertices of tree  $T$ . Letter labels denote sets of leaf neighbours; squares denote invisible vertices.



**Fig. 2.** Clique graph  $G$  ( $k = 4$ ) of tree  $T$  in Figure 1.

A clique graph is an *ideal clique graph* if it can be generated from a  $k$ -leaf power of an ideal tree  $T$ . Ideal clique graphs have elegant properties which are absent from general clique graphs. We can view a general clique graph as having been generated from an ideal clique graph, with subsequent “collapsing” occurring at the invisible nodes.

## 4.2 General Clique Graphs and Clique Graph Partitioning

The presence of invisible nodes complicates the characterization of general clique graphs of 4-leaf powers, for which the correlation between neighbourhoods and levels is no longer so clean. For example, if  $u$ ,  $v$ ,  $w$ , and  $x$  form a path in  $T$  such that  $v$  and  $x$  are invisible and the other neighbours of  $w$  are all leaves, then the 4-neighbourhood of  $w$  is equal to the 2-neighbourhood of  $w$  and the 3-neighbourhoods of  $(v, w)$  and  $(w, x)$ , and is a subset of the 4-neighbourhood of  $v$ . As a consequence of the blurring of distinctions between types of neighbourhoods, intuitively, general clique graphs of 4-leaf powers have the following structure: the sections of height 3 look like ideal clique graphs, but the sections of level 2 can be arbitrary bipartite trees. This is proved in Theorem 1. The following lemma characterizes a few constraints on the correlations between levels and neighbourhoods:

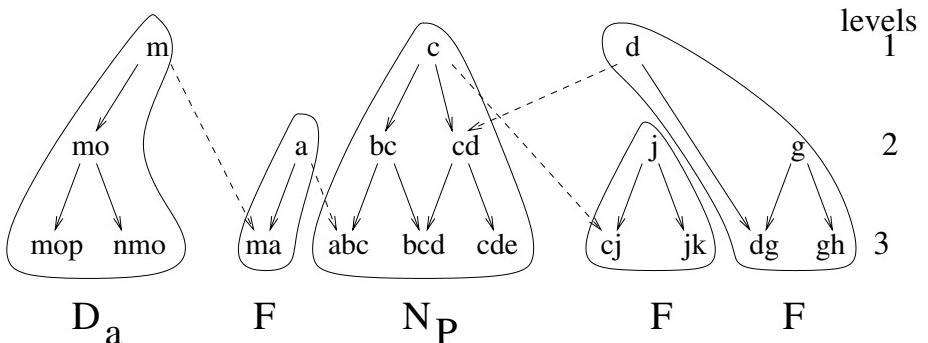
**Lemma 5.** *The clique graph of a 4-leaf power has at most three levels. In any path of length three, the labels of the nodes are, in order from source to*

sink, a 2-neighbourhood, a 3-neighbourhood, and a 4-neighbourhood. A level-3 node is always a 4-neighbourhood, sometimes a 3-neighbourhood, and never a 2-neighbourhood. A level-2 node can be a 2-neighbourhood and/or a 3-neighbourhood, but never a 4-neighbourhood.  $\square$

Given a three-level clique graph  $C$ , we decompose  $C$  into a set of subgraphs and linking edges. We identify two types of three-level subgraphs, namely nondegenerate and degenerate three-level subgraphs, as well as two-level subgraphs. In Theorem II we stipulate additional conditions which ensure that  $C$  is the clique graph of a 4-leaf power.

The decomposition algorithm starts by creating a partition  $\mathcal{N}$  of level-1 nodes that have at least two level-2 children, where two nodes are in the same set of the partition if they share a level-2 child. For each set  $P$  of the partition, it forms the subgraph  $N_P$  of  $C$  induced by  $P$ , the level-2 children of nodes in  $P$ , and the grandchildren of nodes in  $P$ . These are the nondegenerate three-level subgraphs. Removing every  $N_P$  from  $C$  temporarily, the algorithm starts to form the set  $A$  of roots of degenerate three-level subgraphs. While there exists in  $C$  a level-1 node  $a$  with a level-2 child in  $C$ , it forms the subgraph  $D_a$  of  $C$  induced by  $a$ , its level-2 children and its grandchildren.  $D_a$  is temporarily removed from  $C$  and  $a$  is added to  $A$ .

What remains must be two-level subgraphs. The algorithm forms the subgraph  $F$  induced on vertices in  $C$ , renaming each level-1 vertex to be a level-2 vertex, and forms the set  $E$  of linking edges, namely all edges of  $C$  not in any  $N_P$ ,  $D_a$ , or  $F$ . All removed components are restored, and the partitioning is done. We must now reason about its effects, and discover enough structure to justify the reconstruction algorithm. Figure 3 illustrates the decomposition of the clique graph  $G$  from Figure 2, with linking edges appearing as dashed lines.



**Fig. 3.** Partitioned clique graph  $G$ .

Since the label of a node  $a$  of the clique graph  $C$  of a 4-leaf power of a fixed tree  $T$  can be both an  $i$ -neighbourhood and a  $j$ -neighbourhood for  $i \neq j$ , we introduce the notion of a *range* (intuitively, the size of the visible part of

the neighbourhood) and a *middle* (the center of the visible part of the neighbourhood). More formally, the *range* of  $a$  is the length of the longest path in  $T$  connecting leaves in  $\text{cglab}(a)$ . The *middle* of  $a$ , denoted  $\text{middle}(a)$ , is the vertex/edge of  $T$  that is in the middle of such a path.

**Lemma 6.** *If  $c$  is a node of range  $k$ , then  $\text{cglab}(c)$  is the  $k$ -neighbourhood of its middle.*  $\square$

*Proof.* Any vertex in  $\text{cglab}(a)$  is at distance at most  $k/2$  from the middle, otherwise a path of length greater than  $k$  connecting vertices in  $\text{cglab}(a)$  can be constructed. Furthermore, the longest path in an  $i$ -neighbourhood with center vertex/edge  $v$  is of length at most  $i$ . To see this, let the endpoints of the longest path be  $v_1$  and  $v_2$ , and note that  $d(v_1, v_2) \leq d(v_1, v) + d(v, v_2) \leq i$ . Thus  $\text{cglab}(a)$  cannot be an  $i$ -neighbourhood for  $i < k$ , and it must be a  $k$ -neighbourhood.  $\square$

We can extract structure by examining middles in conjunction with levels of nodes.

**Lemma 7.** *For  $c$  a node in a clique graph  $C$  of the 4-leaf power of a tree  $T$ ,*

1. *if  $\text{middle}(c)$  is a vertex  $v$  for a level-3 node  $c$ , then  $\text{cglab}(c)$  is the 4-neighbourhood of  $v$  and  $v$  has at least two visible neighbours;*
2. *if  $\text{middle}(c)$  is a vertex  $v$  for a level-2 node  $c$ , then  $\text{cglab}(c)$  is the 2-neighbourhood of  $v$  and  $v$  is visible; and*
3. *if  $\text{middle}(c)$  is an edge  $e$ , then  $\text{cglab}(c)$  is the 3-neighbourhood of  $e$  and both endpoints of  $e$  are visible.*  $\square$

We are now ready for the main theorem on decompositions of clique graphs.

**Theorem 1.** *The following conditions are true of the clique graph  $C$  of the connected 4-leaf power  $G$  of a tree  $T$ :*

1. *For  $P_1$  and  $P_2$  distinct sets in the partition  $\mathcal{N}$ ,  $N_{P_1}$  and  $N_{P_2}$  do not intersect.*
2. *For  $a_1$  and  $a_2$  distinct vertices in  $A$ ,  $D_{a_1}$  and  $D_{a_2}$  do not intersect.*
3. *For any  $P \in \mathcal{N}$  and  $a \in A$ ,  $N_P$  and  $D_a$  do not overlap.*
4. *Each  $N_P$  is isomorphic to a (necessarily ideal) clique graph of an ideal subtree.*
5. *In each  $D_a$ ,  $a$  has only one child and exactly two grandchildren.*
6.  *$F$  is a forest without edges connecting nodes of the same level.*
7. *A linking edge can connect either a level-1 node of a three-level subgraph and a level-3 node of two-level subgraph (central linking edge), a level-2 node (formerly level-1) of a two-level subgraph and a level-2 node of a three-level subgraph (peripheral linking edge), or a level-2 node of a two-level subgraph and a level-3 node of a three-level subgraph (marginal linking edge).*

*Proof.* (sketch) (1) and (5) follow with a bit of reasoning about properties of clique graphs already proven. (2) and (3) are direct consequences of the algorithm. The hardest parts to prove are (4) and (6). (4) is proved by identifying

an ideal subtree of  $T$  and then proving that its clique subgraph is isomorphic to the component  $N_P$ ; (6) is proved by showing that  $C_{2,3}$ , which contains  $F$  as a subgraph, is topologically equivalent to a forest  $F^*$  created from  $T$ . These last two proofs occupy several typeset pages and make good use of Lemma 7. Finally, (7) is proved using tools developed in the proof of (4).  $\square$

Referring to vertices by their clique graph labels, in Figure 3, the edges from  $m$  to  $ma$  and from  $c$  to  $cj$  are central linking edges, the edge from  $d$  to  $cd$  is a peripheral linking edge, and the edge from  $a$  to  $abc$  is a marginal linking edge.

Finally, we can quantify the constants in the linear bounds on the number of nodes and edges in the clique graph of a 4-leaf power.

**Lemma 8.** *If  $G$  is the connected 4-leaf power of a tree  $T$ , then the clique graph algorithm generates at most  $6n$  nodes and at most  $18n$  edges.*

*Proof.* (sketch)  $T$  has at most  $2n$  internal vertices since no two invisible vertices can be adjacent (otherwise the 4-leaf power is disconnected) and every visible vertex has at least one associated leaf. Since each node in the clique graph is a 2-, 3-, or 4-neighbourhood of an internal vertex of  $G$ , there are at most  $6n$  possible nodes in the clique graph. Each edge between a level-1 node and a level-2 node is either part of a three-level subgraph or is a peripheral linking edge, so we can prove that these form a tree and hence number at most  $6n$ . Similarly, the edges between level-2 and level-3 nodes number at most  $6n$ .

We finally determine the number of edges that may be generated between level-1 and level-3 nodes (some of which will be subsequently deleted in the transitive reduction). There is an edge from a level-1 node (a 2-neighbourhood) to each 4-neighbourhood containing it. Since the 2-neighbourhood of a vertex  $v$  is contained in exactly the 4-neighbourhoods of  $v$  itself and all of  $v$ 's neighbours, the total number of edges is the sum over all internal vertices of the degree of the vertex plus one. This sum equals the number of internal vertices plus twice the number of nonpendant edges in  $T$ , and the total is at most  $6n$ .  $\square$

## 5 Reconstructing the Underlying Tree of a 4-Leaf Power

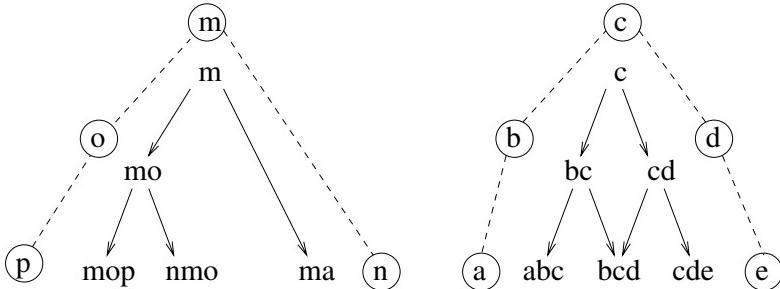
We will briefly sketch the intuition behind our reconstruction algorithms before giving details. For  $k = 3$ , our assumption that  $G$  is connected makes its clique graph ideal. As a result, simple local replacement in the clique graph will construct a suitable tree. Due to space reasons, we omit the algorithm and justification, which is also simple. For  $k = 4$ , things are not so simple. Each subgraph of the partitioned clique graph is treated by an appropriate form of local replacement, and the trees thus obtained are joined in a suitable manner. For clarity, we refer to *nodes* in the clique graph and *vertices* in the created tree  $T$ . In the course of the algorithm, vertices are labeled with subsets of  $V(G)$ .

The algorithm first creates a partitioned clique graph. For each three-level nondegenerate component  $N$ , it checks for the the following properties, which (by an omitted technical lemma) must be true of any ideal clique graph of height

3. Every level-1 node must have at least two children, and all its children must have a unique common level-3 child. Every level-2 node must have exactly two children and if it has two parents, its label is the union of theirs. Two or more parents of a level-3 node must have a unique common parent.

If these conditions are satisfied, the algorithm creates a subtree  $T_N$  for each nondegenerate three-level component  $N$  as described below.  $T_N$  is initially empty. For each level-1 node  $a$  in  $N$ , it creates a vertex  $t(a)$  labeled  $\text{cglabel}(a)$ . If level-1 nodes  $a$  and  $b$  share a child, it creates the edge  $(t(a), t(b))$ . For each border node  $a$  with parent  $b$ , it creates a vertex  $t(a)$  in  $T_N$  labeled  $\text{cglabel}(a) \setminus \text{cglabel}(b)$ , and the edge  $(t(a), t(b))$ . For each level-3 node  $a$  with two parents  $b$  and  $c$  such that  $A = \text{cglabel}(a) \setminus (\text{cglabel}(b) \cup \text{cglabel}(c))$  is nonempty, it creates a vertex  $t(a)$  labeled  $A$ , and for  $d$  the common parent of  $b$  and  $c$ , it creates edge  $(t(a), t(d))$ .

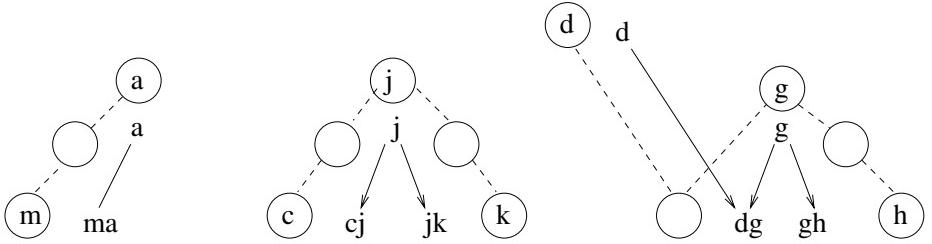
Next each degenerate three-level component  $D$  is checked to ensure that it is a degenerate ideal clique graph (one level-1 and one level-2 node, at most two level-3 nodes), and from it a tree  $T_D$  is formed as described below. For the level-1 node  $a$ , the algorithm creates  $t(a)$  labeled  $\text{cglabel}(a)$ ; for the level-2 node  $b$ , it creates  $t(b)$  labeled  $\text{cglabel}(b) \setminus \text{cglabel}(a)$ ; and for the level-3 nodes  $c$  and  $d$ , it creates  $t(c)$  labeled  $\text{cglabel}(c) \setminus \text{cglabel}(b)$  and  $t(d)$  labeled  $\text{cglabel}(d) \setminus \text{cglabel}(b)$ , as well as edges  $(t(d), t(a)), (t(a), t(b)), (t(b), t(c))$ . Figure 4 illustrates the subtrees derived for the three-level components of Figure 3.



**Fig. 4.** Subtree derived from three-level components of Figure 3.

The subgraph  $F$  induced on nodes in  $C$  not in any  $N$  or  $D$  must be a forest of nodes at levels 2 and 3. If it is, subtrees of  $T$  are created from its components. For each tree  $S$  in  $F$ , an initially empty subtree  $T_S$  is created. For each level-2 node  $a$  in  $S$ , the algorithm creates a vertex  $t(a)$  labeled  $\text{cglabel}(a)$ . For each level-3 node  $a$ , if  $A = \text{cglabel}(a) \setminus \cup_b$  parent of  $a$   $\text{cglabel}(b)$  is empty, it creates a vertex  $t(a)$  with the empty label, and an edge  $(t(a), t(b))$  for each parent  $b$  of  $a$ . Otherwise, it creates a vertex  $t(a)$  labeled  $A$ , a vertex  $v_a$  with the empty label, the edge  $(t(a), v_a)$ , and an edge  $(t(b), v_a)$  for each parent  $b$  of  $a$ . Figure 5 illustrates this process.

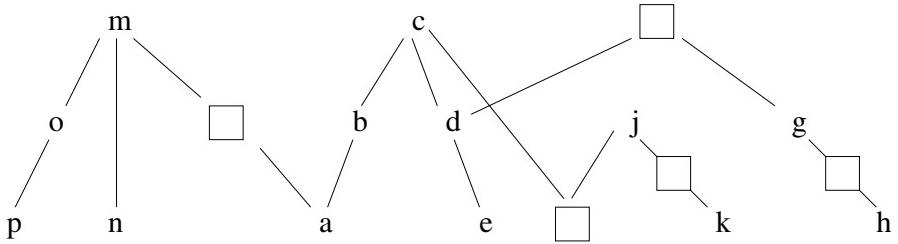
The union of all  $T_N$ ,  $T_D$ , and  $T_S$  forms a labeled forest  $L$ . Subtrees are connected as specified by linking edges. For each central linking edge  $(a, b)$ ,  $a$  in



**Fig. 5.** Subtrees derived from two-level components of Figure 2.

$T_N$  or  $T_D$ ,  $b$  in  $T_S$ , it creates the edge  $(t(a), v_b)$ , and removes the label of  $t(a)$  from the label of  $t(b)$ . For each peripheral or marginal linking edge  $(a, b)$ ,  $a$  in  $T_N$  or  $T_D$ ,  $b$  in  $T_S$ , it identifies the nodes  $t(a)$  and  $t(b)$ .

Finally, the vertex label of each vertex  $v$  is replaced by a set of leaves with those names adjacent to  $v$ . Figure 6 shows the reconstructed tree for the running example. Although it is not identical to Figure 1 (as a clique graph can represent more than one possible tree), it differs only in the absence of invisible vertices between  $g$  and  $h$  and between  $j$  and  $k$ .



**Fig. 6.** Tree generated from clique graph of Figure 2 by reconstruction algorithm. As before, sets of leaf neighbours are indicated by letter labels, and invisible vertices by squares.

The correctness of the algorithm follows from the two lemmas below. The first shows that the 4-leaf power of the constructed tree  $T$  is a subgraph of  $G$ . The second shows that  $G$  is a subgraph of the 4-leaf power of  $T$ .

**Lemma 9.** *If leaves  $u$  and  $v$  are of distance at most four in  $T$ , then there exists an edge  $(u, v)$  in  $G$ .*

*Proof.* (sketch) In the tree  $T$  formed by the algorithm, we consider all possible parents  $p$  and  $q$  of  $u$  and  $v$  such that the distance between  $p$  and  $q$  is at most two, and show that in each case  $(u, v)$  must have been an edge in  $G$ .  $\square$

**Lemma 10.** *If  $(u, v)$  is an edge in  $G$ , then  $u$  and  $v$  are of distance at most four in  $T$ .*

*Proof.* (sketch) Since  $u$  and  $v$  are leaves of  $T$ , it suffices to show that their associated internal vertices are of distance at most two in  $T$ . The edge  $(u, v)$  must be in some maximal clique of  $G$ , and so  $u$  and  $v$  appear together in some label of a level-3 node  $a$  in  $C$ . The proof proceeds by looking at what happens to node  $a$  during the algorithm, and in which vertex labels  $u$  and  $v$  can be found.  $\square$

**Theorem 2.** *Given a graph  $G$  with  $n$  vertices and  $e$  edges, it is possible in time  $O(n^3)$  to determine whether or not  $G$  is a 4-leaf power or a 3-leaf power of a tree  $T$ , and if so, to determine such a  $T$ .*

*Proof.* (sketch) We have seen that clique graph generation takes  $O(n^3)$  time, and produces a clique graph with  $O(n)$  vertices and edges; partitioning and local replacement then clearly take time  $O(n)$ .  $\square$

## 6 Conclusions and Further Work

Reconstructing the  $k$ -leaf powers of ideal trees for  $k > 4$  would be easy (by generalizing the part of the reconstruction algorithm that deals with nondegenerate three-level subgraphs) but less interesting than handling more general trees. We believe the clique graph approach offers promise for the general case, though more work is needed to quantify exactly how collapses occur as a result of invisible vertices. The main stumbling block appears to be the combinatorial explosion in the number of cases in the analysis of the extension of results like Theorem 11, which may be controlled by discovery of further general structure. It might also be possible to extend these techniques to consider the case of weighted edges in the tree  $T$ .

Among the objections to practical use of the algorithms is that the number of trees that correspond to a particular  $k$ -leaf power could be very large. It might be interesting to determine all corresponding trees, or perhaps all trees that satisfy a given set of additional constraints.

### Acknowledgements

We would like to thank Paul Kearney for suggesting this problem to us. We are grateful to the anonymous referees whose helpful comments improved the presentation of this paper.

### References

- BG91. J.-P. Barthélemy and A. Guénoche. *Trees and Proximity Representations*. John Wiley and Sons, 1991.
- CK98. D. G. Corneil and P. Kearney. Tree powers. *Journal of Algorithms*, 29:111–131, 1998.

- Gav72. R. Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set for chordal graphs. *SIAM Journal on Computing*, 1:180–187, 1972.
- HKW99. M. R. Henzinger, V. King, and T. Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24(1):1–13, 1999.
- HSW99. D. H. Huson, K. A. Smith, and T. Warnow. Estimating large distances in phylogenetic reconstruction. *Algorithm Engineering*, pages 270–285, 1999.
- KW99. J. Kim and T. Warnow. Tutorial on phylogenetic tree estimation. manuscript, Department of Ecology and Evolutionary Biology, Yale University, <http://ismb99.gmd.de/TUTORIALS/Kim/4KimTutorial.ps>, 1999.
- Lin92. N. Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21:193–201, 1992.
- MS94. R. Motwani and M. Sudan. Computing roots of graphs is hard. *Discrete Applied Mathematics*, 54:81–88, 1994.
- RTL76. D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.
- TY84. R. E. Tarjan and M. Yannakakis. Simple linear-time algoritlms to test chodality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13:566–579, 1984.

# Recognizing Weakly Triangulated Graphs by Edge Separability

Anne Berry<sup>1</sup>, Jean-Paul Bordat<sup>1</sup>, and Pinar Heggernes<sup>2</sup>

<sup>1</sup> LIRMM, 161 Rue Ada, F-34392 Montpellier, France  
`{aberry, bordat}@lirmm.fr`

<sup>2</sup> Department of Informatics, University of Bergen, N-5020 Bergen, Norway  
`pinar@ii.uib.no`

**Abstract.** We apply Lekkerkerker and Boland’s recognition algorithm for triangulated graphs to the class of weakly triangulated graphs. This yields a new characterization of weakly triangulated graphs, as well as a new recognition algorithm which, unlike the previous ones, is not based on the notion of 2-pair, but rather on the structural properties of the minimal separators of the graph. It also gives the strongest relationship to the class of triangulated graphs that has been established so far.

## 1 Introduction

Weakly triangulated graphs were introduced by Hayward [10] as a natural extension of the perfect class of triangulated graphs. A graph is *triangulated*, or *chordal*, if it does not contain a chordless cycle on four or more vertices. A graph is *weakly triangulated* if neither the graph nor its complement contains a chordless cycle on five or more vertices, or equivalently, the graph contains neither a *hole* nor an *antihole*. A graph with no hole can fail to be perfect, but Hayward proved that for weakly triangulated graphs perfection is preserved.

This class has given rise to a continuous flow of research [15], [22], [13], [14], [3], [23], [7], [16]. In particular, time complexity for recognition of the class has steadily improved over the years. Hayward [11] proposed an  $O(n^5)$  recognition algorithm for weakly triangulated graphs that checks for the presence of a hole in the graph and then in its complement. This was improved to  $O(n^{4.376})$  by Spinrad’s hole-finding procedure [21].

Hayward, Hoàng, and Maffray [15] characterized weakly triangulated graphs by the presence of a *2-pair*: a pair  $\{a, b\}$  of non-adjacent vertices such that every chordless path from  $a$  to  $b$  has exactly two edges. Arikati and Rangan [1] gave an efficient algorithm for finding a 2-pair, and Spinrad and Sritharan [22] used this to improve the recognition to  $O(n^2m)$  by repeatedly finding a 2-pair  $\{a, b\}$  and adding the edge  $ab$ , until the graph becomes complete. Their idea is that adding an edge between the vertices that make up a 2-pair preserves the property of being weakly triangulated, because if  $\{a, b\}$  is a 2-pair, then  $a$  and  $b$  together cannot belong to a hole or an antihole. Note that this also implies that an edge  $ab$  which is a 2-pair of the complement graph (called a *co-pair* [16]) can likewise be deleted without changing the property of being weakly triangulated.

Hayward [13], [14], showed the presence of a separable set of *edges*, called a *handle*, which form a connected subset such that all the edges of the handle see all the vertices of the corresponding separator. This notion was used recently by Hayward, Spinrad, and Sritharan [16] to give an  $O(m^2)$  recognition algorithm, which finds a set of co-pairs by computing a handle of a handle recursively, and repeatedly removing a co-pair, until no edge is left in the graph.

In this paper, we introduce an algorithm for recognition of weakly triangulated graphs that is not based on the notion of 2-pair or co-pair. Several attempts have been made to establish a structural relationship between triangulated graphs and weakly triangulated graphs. We establish a strong connection between these two classes of graphs using the notion of *LB-simpliciality*, which we will define in Section 3. The proof of this new structural relationship and a recognition algorithm based on this result are given in Section 4. This introduces a totally new and different approach to the recognition of weakly triangulated graphs. We define LB-simpliciality for edges, and we simply check that all the edges of the given graph are LB-simplicial. Although we believe that our algorithm can be implemented to match the time bound  $O(m^2)$ , the proven time complexity is  $O(n^2m)$ . Among the strengths of the presented algorithm are its simplicity and elegance. In addition, the algorithm is highly parallel in spirit. The edges of the given graph can be checked for the desired property independently of each other in any order, or in particular in parallel. None of the previous algorithms for recognition of weakly triangulated graphs have this property.

## 2 Preliminaries

All graphs in this work are undirected and finite. A graph is denoted  $G = (V, E)$ , with  $n = |V|$ , and  $m = |E|$ .  $G(A)$  denotes the subgraph induced by a vertex set  $A \subset V$ , but we will often denote it simply by  $A$  when there is no ambiguity;  $\overline{G}(A)$  denotes the subgraph induced by  $A$  in the complement of  $G$ .

A graph is *complete* if all of its vertices are pairwise adjacent. A *clique* in a graph is a complete subgraph. We denote a path on  $k$  vertices by  $P_k$ . A *chord* is an edge between two non-consecutive vertices of a path or a cycle. A *hole* is an induced chordless cycle on five or more vertices, and an *antihole* is the complement of a hole. In this paper, we will regard all subgraphs as vertex sets.

The *neighborhood* of a vertex  $x$  is  $N(x) = \{y \neq x \mid xy \in E\}$ ; we will say that a vertex  $x$  *sees* another vertex  $y$  iff  $xy \in E$ . The neighborhood of a set of vertices  $A$  is  $N(A) = \cup_{x \in A} N(x) - A$ . A vertex is *simplicial* if its neighborhood is a clique. For a set of vertices  $A$ , a *confluence point* is a vertex of  $A$  that sees all the vertices in  $N(A)$ .

In order to have analogous definitions for edges, we regard an edge  $ab$  as a set of vertices  $\{a, b\}$ . The neighborhood of an edge is a vertex set. We will let  $N(ab)$  denote the neighborhood of edge  $ab$ , i.e.  $N(\{a, b\})$ . Hence, an edge  $ab$  sees a vertex  $x$  if either  $a$  or  $b$  (or both) sees  $x$ .

For  $X \subseteq V$ ,  $C(X)$  denotes the set of connected components of  $G(V - X)$  (connected components are also vertex sets).  $S \subset V$  is called a *separator* if

$|C(S)| \geq 2$ , an *ab-separator* if  $a$  and  $b$  are in different connected components of  $C(S)$ , a *minimal ab-separator* if  $S$  is an *ab-separator* and no proper subset of  $S$  is an *ab-separator*, and a *minimal separator* if there is some pair  $\{a, b\}$  such that  $S$  is a minimal *ab-separator*. Equivalently,  $S$  is a *minimal separator* if there exist  $C_1$  and  $C_2$  in  $C(S)$  such that  $N(C_1) = N(C_2) = S$ . A component  $C$  of  $C(S)$  is called *full* if  $N(C) = S$ .  $S(G)$  denotes the set of minimal separators of  $G$ . A set  $A \subset V$  is *separable* if  $N(A)$  is a minimal separator.

### 3 Connections between Triangulated and Weakly Triangulated Graphs

The notion of a *simplicial vertex* in a triangulated graph was introduced independently by Dirac [8] and by Lekkerkerker and Boland [18] as an extension of the notion of a leaf in a tree, and is the basis for the following theorem by Dirac:

**Theorem 1.** [8] *Any non-complete triangulated graph has at least two non-adjacent simplicial vertices.*

This led Fulkerson and Gross [9] to define their famous and characterizing simplicial elimination scheme:

**Characterization 1.** [9] *A graph is triangulated iff one can repeatedly find a simplicial vertex and delete it from the graph, until no vertex is left.*

Hayward [12] proposed a construction scheme for weakly triangulated graphs which is inspired by this elimination scheme. He notes the following: 1. Triangulated graphs can be generated by repeatedly adding a vertex which is not the middle vertex of a  $P_3$ ; this added vertex is precisely a simplicial vertex. 2. Likewise, weakly triangulated graphs can be generated by repeatedly adding an *edge* which is not the middle edge of a  $P_4$ . This result shows that an *edge* in a weakly triangulated graph plays a role similar to the one a *vertex* plays in a triangulated graph.

Another interesting property of the same essence was suggested by Kratsch [17]. In a triangulated graph, for every minimal separator  $S$ , every component  $C$  of  $G(V - S)$  contains a confluence point. Kratsch showed that in a weakly triangulated graph, for every minimal separator  $S$ , every full component  $C$  contains either a confluence point or a confluence edge, i.e. an edge  $e$  such that  $N(C) \subseteq N(e)$ . Independently of this result, Hayward [13] introduced the stronger notion of  $S$ -saturating edge, which will be defined in Section 4.

Our contribution in this paper is the extension of a characterization of triangulated graphs due to Lekkerkerker and Boland [18], which implicitly uses separation. In a paper contemporary to Dirac's, they show that interval graphs are the graphs that are both triangulated and AT-free (i.e. devoid of asteroidal triples). Their search for an efficient recognition algorithm led them to study triangulated graphs and to propose a characterization for these, which, with the hindsight we now have on minimal separation, can be expressed in the following fashion:

**Characterization 2.** [18] *A graph is triangulated iff for every vertex  $x$ , all the minimal separators included in  $N(x)$  are cliques.*

In order to simplify notations, we use the following definition (the abbreviation LB refers to Lekkerkerker-Boland in all contexts throughout the paper):

**Definition 1.** *A vertex is LB-simplicial if all the minimal separators included in its neighborhood are cliques.*

Characterization 2 can thus be reformulated in the following way:

**Characterization 3.** *A graph is triangulated iff every vertex is LB-simplicial.*

Linear-time algorithms for the recognition of triangulated graphs are based on Characterization 1, as they require computing a simplicial ordering, which was first done efficiently by the famous algorithm known as LexBFS, due to Rose, Tarjan, and Lueker [20].

Recently, Berry, Bordat, and Heggernes [2], [6], used Characterization 3 to compute a minimal triangulation of a graph by checking (in an arbitrary order) the vertices for LB-simpliciality, and adding the necessary edges whenever an anomaly is detected.

In this paper, we extend the notion of *LB-simplicial vertex* to that of *LB-simplicial edge*, and show how we can derive an elegant and straightforward recognition algorithm for weakly triangulated graphs by simply checking each edge for LB-simpliciality. Thus, by extending Characterization 3, we will prove in the next section that a graph is weakly triangulated iff every edge is LB-simplicial.

## 4 Weakly Triangulated Graph Recognition

In this section, we extend Lekkerkerker and Boland's algorithm for the recognition of triangulated graphs to weakly triangulated graphs.

### 4.1 Lekkerkerker and Boland's Algorithm for Triangulated Graph Recognition

Translated into our terminology, Lekkerkerker and Boland's algorithm is the following:

**Algorithm LB-TG**

```

input : A graph  $G = (V, E)$ , with  $|V| = n$  and  $|E| = m$ .
output : An answer to the question: “Is  $G$  triangulated?”
begin
  foreach  $v \in V$  do
    if  $v$  is not an LB-simplicial vertex then
      return( $G$  is not triangulated);
  return( $G$  is triangulated);
end

```

Checking for LB-simpliciality of a vertex  $x$  requires computing the set of minimal separators included in the neighborhood of  $x$ . In general, generating minimal separators can be done by computing the neighborhoods of the connected components resulting from the removal of certain vertex sets [5], [23]. In [18] the minimal separators in the neighborhood of vertex  $x$  are computed in the following way: for each component  $C$  in  $C(x \cup N(x))$ , compute  $N(C)$ , which is a minimal separator included in  $N(x)$ . With the following theorem, we give a formal description of all the minimal separators included in a clique neighborhood.

**Theorem 2.** *Let  $K$  be a clique of a graph  $G$ . The set of minimal separators included in  $N(K)$  is exactly  $M = \{N(C) \mid C \in C(K \cup N(K))\}$ .*

*Proof.* For each  $C \in C(K \cup N(K))$ ,  $N(C)$  is a separator that separates  $C$  from  $K$ . Thus  $G(V - N(C))$  has at least two components; one is  $C$ , and another is the one containing  $K$ . Since  $N(C) \subseteq N(K)$ , both these components have the same neighborhood,  $N(C)$ , and consequently  $N(C)$  is a minimal separator. We have to also show that there are no minimal separators included in  $N(K)$  outside of  $M$ . Assume that  $S \subseteq N(K)$  is a minimal separator. Then there must exist at least two components  $C_1$  and  $C_2$  in  $G(V - S)$  such that  $N(C_1) = N(C_2) = S$  (i.e. full components). Since  $K$  is a clique, and  $S \subseteq N(K)$ , the whole of  $K$  must be included in some full component. Let  $C_1$  be a full component *not* containing  $K$ . Since  $C_1$  cannot contain any neighbor of  $K$ ,  $C_1$  must belong to  $C(K \cup N(K))$  and the proof is complete.

## 4.2 A Characterization of Weakly Triangulated Graphs by LB-Simplicial Edges

Our approach is based on Theorem 1 from Hayward’s original paper [10], which we express as:

**Theorem 3.** [10] *Let  $G$  be a weakly triangulated graph, and let  $S$  be a minimal separator of  $G$  such that  $\overline{G}(S)$  is connected. Then in each full component  $C$  of  $C(S)$ , there is a vertex that sees all the vertices of  $S$ .*

Hayward in [13] derives the following concept:

**Definition 2.** [13] Given a set  $S$  of vertices, an edge  $e$  of  $G(V - S)$  is said to be  $S$ -saturating if, for each component  $S_j$  of  $\overline{G}(S)$ , at least one endpoint of  $e$  sees all vertices of  $S_j$ .

and shows that in each full component of a minimal separator in a weakly triangulated graph, there is either a confluence point or an  $S$ -saturating edge.

The following definition is central for the main result of this paper. We define an *LB-simplicial edge* based on the role such an edge plays in a weakly triangulated graph. The importance of this notion for weakly triangulated graph recognition is analogous to that of an LB-simplicial vertex for triangulated graphs.

**Definition 3.** An edge  $e$  of  $E$  is LB-simplicial if, for each minimal separator  $S$  included in the neighborhood of  $e$ ,  $e$  is  $S$ -saturating.

We will quite naturally consider as LB-simplicial an edge  $e$  such that  $e \cup N(e) = V$ .

According to Theorem 2, the set of minimal separators included in the neighborhood of an edge  $e$  can be computed in the following fashion: for each component  $C$  of  $C(e \cup N(e))$ , compute  $N(C)$ .

**Theorem 4.** (Main Theorem) A graph  $G = (V, E)$  is weakly triangulated iff every edge of  $E$  is LB-simplicial.

*Proof.* We will also prove a slightly stronger property, namely that an LB-simplicial edge cannot belong to a hole.

$\Leftarrow$  Let  $G$  be a graph in which every edge is LB-simplicial.

1. Suppose that  $G$  has a hole  $x_1x_2...x_k$ ,  $k \geq 5$ . Clearly,  $x_4, \dots, x_{k-1}$  belong to the same component  $C$  of  $C(x_1x_2 \cup N(x_1x_2))$ , and  $x_3$  and  $x_k$  belong to the same connected component  $S_1$  of  $\overline{G}(N(C))$ , where  $N(C)$  is a minimal separator and a subset of  $N(x_1x_2)$ . As  $x_2$  fails to see  $x_k$  and  $x_1$  fails to see  $x_3$ , edge  $x_1x_2$  cannot be  $N(C)$ -saturating, which contradicts the assumption that  $x_1x_2$  is an LB-simplicial edge. Note that this argument holds for any edge of a hole, thus no edge of a hole can be LB-simplicial.
2. Suppose that  $G$  has an antihole which is the complement of a hole on  $x_1x_2...x_k$ ,  $k > 5$ , (for  $k = 5$ ,  $x_1x_2...x_5$  is also a hole). Thus  $x_2x_k$  is an edge that fails to see  $x_1$ . Let  $C$  be the component of  $C(x_2x_k \cup N(x_2x_k))$  containing  $x_1$ , so that  $N(C)$  is a minimal separator included in  $N(x_2x_k)$ . Vertices  $x_3, \dots, x_{k-1}$  are all in the neighborhood of both  $x_1$  and  $x_2x_k$ , thus they all belong to  $N(C)$ . Clearly,  $x_3, \dots, x_{k-1}$  belong to the same connected component  $S_1$  of  $\overline{G}(N(C))$ . But  $x_k$  fails to see  $x_{k-1}$  and  $x_2$  fails to see  $x_3$ , thus edge  $x_2x_k$  is not  $N(C)$ -saturating, and fails to be LB-simplicial.

$\Rightarrow$  Let  $G$  be a weakly triangulated graph, and suppose some edge  $ab$  fails to be LB-simplicial. Let  $S = N(C)$  be a minimal separator contained in the neighborhood of  $ab$  for which  $ab$  fails to be  $S$ -saturating, let  $S_1$  be a connected

component of  $\overline{G}(S)$  such that neither  $a$  nor  $b$  sees all the vertices of  $S_1$ , and consider the subgraph  $G'$  induced by  $C \cup S_1 \cup ab$ . As any subgraph of a weakly triangulated graph is itself weakly triangulated,  $G'$  must be weakly triangulated.  $S_1$  is a minimal separator of  $G'$ , with 2 full components,  $\{a, b\}$  and  $C$ .  $\overline{G}'(S_1)$  is connected, and for which  $ab$  is the only edge. Neither  $a$  nor  $b$  sees all the vertices of  $S_1$ , which contradicts Theorem 3.

### 4.3 Recognition Algorithm

Theorem 4 yields a recognition algorithm as direct application:

#### Algorithm LB-WT

```

input : A graph  $G = (V, E)$ , with  $|V| = n$  and  $|E| = m$ .
output : An answer to the question: “Is  $G$  weakly triangulated”, and if  $G$  is
not, an edge  $e$  that belongs to a hole or an antihole.

begin
  foreach  $e \in E$  do
    if  $e$  is not an LB-simplicial edge then
      return( $G$  is not weakly triangulated, and  $e$  belongs to a hole or
      an antihole);
    return( $G$  is weakly triangulated);
  end
```

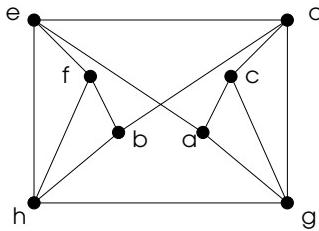
*Remark 1.* In a weakly triangulated graph, minimal separators which are *not* in any edge neighborhood are an exception; this means that every component in  $C(S)$  is restricted to a single (confluent) vertex, thus the graph would be of diameter two. Consequently, just as in a triangulated graph every minimal separator is included in some vertex neighborhood, in a non-trivial weakly triangulated graph, every minimal separator is included in some edge neighborhood. Thus LB-type algorithms actually scan the whole set of minimal separators and test them. Just as minimal separators in a triangulated graph are characterized as cliques included in the neighborhood of a vertex, the minimal separators of a weakly triangulated graph can be characterized by the LB-simplicial edges whose neighborhood contains them [4].

*Example 1.* In Figure 11, we use the first example from Hayward’s original paper [10]. This graph is weakly triangulated, isomorphic to its complement, and devoid of clique separators, and thus of simplicial and co-simplicial vertices. We will only demonstrate LB-simplicity of one edge.

LB-simplicity testing of edge  $bh$ :  $N(bh) = \{d, e, f, g\}$ , and  $C(N(bh) \cup bh) = \{a, c\}$ . The only minimal separator of  $G$  included in the neighborhood of  $bh$  is

$N(\{a, c\}) = \{d, e, g\}$ . Connected components of  $\overline{G}(\{d, e, g\})$  are  $\{d\}$  and  $\{e, g\}$ . Vertex  $h$  sees both vertices in  $\{e, g\}$ , and  $b$  sees  $d$ . Hence  $bh$  is  $\{d, e, g\}$ -saturating and thus LB-simplicial.

Note that  $de \cup N(de) = V$ , thus edge  $de$  will generate no minimal separator. The total set of minimal separators is  $S(G) = \{\{a, d, g\}, \{a, d, h\}, \{b, e, g\}, \{b, e, h\}, \{c, e, g\}, \{d, e, g\}, \{d, e, h\}, \{d, f, h\}\}$ .



**Fig. 1.** A weakly triangulated graph.

*Remark 2.* Note that, in Algorithm LB-WT, the edges are processed in an arbitrary order. We would thus like to draw the reader's attention to the parallel spirit of this algorithm. Since no edges are added, and the LB-simpliciality of an edge does not depend on that of any other edge, LB-simpliciality testing for all edges can be done in parallel. For a shared memory parallelization, each processor that becomes idle picks an unprocessed edge from the global queue of edges, and checks whether this edge is LB-simplicial. Every processor is able to read the graph which is stored globally, whereas each processor locally computes the components of  $G(V - e \cup N(e))$  and all the information that is necessary to establish the LB-simpliciality of the edge being checked. None of the previous recognition algorithms have the property that edges or vertices of the graph can be processed independently and in an arbitrary order. Moreover, the straightforward parallel implementation described here can be enhanced along the guidelines presented in the complexity analysis given in the next subsection.

#### 4.4 Complexity

For each of the  $m$  edges  $ab$  in  $E$ , computing  $ab \cup N(ab)$  requires  $O(n)$  time. Computing the minimal separators contained in the neighborhood of  $ab$  requires computing the connected components of  $C(ab \cup N(ab))$  as well as their neighborhoods in  $G$ , according to Theorem 2, and this can be done in a single  $O(m)$ -time graph search for each edge  $ab$ . We will encounter at most  $n$  minimal separators included in the neighborhood of  $ab$ . Moreover, the sum of the number of vertices in these separators will be less than  $m$  for each edge  $ab$ . Thus for each edge  $ab$ ,

encountering the minimal separators included in  $N(ab)$  can be done in  $O(m+n)$  time, and at most  $O(n)$  separators will be encountered.

One problem is that we may encounter the same separator many times, and we do not want to keep several copies of the same separator. Since each edge encounters at most  $n$  separators, we might have a total of  $mn$  separators if we allow multiple copies. However, it is shown in [4] that the number of minimal separators in a weakly triangulated graph is at most  $m+n$ . In order to avoid multiple copies, we use a suitable data structure to memorize the minimal separators and their co-connected components. Such a structure is described by Nourine and Raynaud [19] (see also [5]). It guarantees that, if the number of separators kept in the structure is  $O(m)$ , then in  $O(n)$  time, we can check whether each newly encountered separator is already in the structure, and if not, insert it in the structure.

Let us look in more detail into how the LB-simpliciality testing of an edge  $e = ab$  can be done:

```

foreach  $x \in N(e)$  do
  if  $x$  sees only  $a$  then  $l(x) = 1$ ;
  if  $x$  sees only  $b$  then  $l(x) = 2$ ;
  else  $l(x) = 3$ ;
foreach  $S \subseteq N(e)$  do
  if  $S$  is not yet in  $S(G)$  then
    insert  $S$  in  $S(G)$ ;
    compute the set of connected components of  $\overline{G}(S)$  and insert them in
    the data structure;
  foreach component  $S_j$  of  $\overline{G}(S)$  do
    if  $\exists\{x, y\} \in S_j \mid l(x) = 1$  and  $l(y) = 2$  then
      return( $e$  is not LB-simplicial);
return( $e$  is LB-simplicial);

```

The first **foreach** loop requires  $O(n)$  time. For the second **foreach** loop, because a weakly triangulated graph has at most  $n+m$  minimal separators, the outer loop must be terminated if the number of minimal separators stocked in the data structure exceeds  $n+m$ . In this case, we can readily conclude that the given graph is not weakly triangulated. In any case, each outer loop has at most  $O(n)$  iterations.

Assuming the above mentioned restriction, processing each minimal separator  $S$  requires:

1. – If  $S$  is not in  $S(G)$ : a search and an insertion:  $O(n)$  per separator, and then the computation of the set of co-connected components:  $O(m)$  per

separator. These operations are only done once per separator, which ensures a global complexity of  $O(m^2)$  for this.

- If  $S$  is in  $S(G)$ : a search and retrieval of the set of co-connected components from the data structure:  $O(n)$ . These operations may have to be done several times for each separator, thus this takes  $O(n^2)$  for each edge since there are at most  $O(n)$  separators in  $N(e)$  (or, equivalently,  $O(n)$  iterations in the outer loop).
- 2. In either case we must check whether edge  $e$  is  $S$ -saturating:  $O(\Sigma|S|, S \subseteq N(e))$ , i.e.  $O(m)$  for each edge  $e$ .

The global time complexity is thus  $O(n^2m)$ , since there are  $m$  steps in Algorithm LB-WT each corresponding to an edge of the given graph. Note that only the second part of Case 1 mentioned above requires  $O(n^2)$  time for each edge, which leads us to conjecture that an amortized complexity analysis would yield a global  $O(m^2)$  time complexity.

## 5 Conclusion

We have shown new structural properties for the class of weakly triangulated graphs, and we have established a strong relationship between this class and triangulated graphs. Based on this novel insight, we have introduced a new recognition algorithm for weakly triangulated graphs, which is easy to follow and understand, and which does not use any of the previously introduced techniques for recognition. Though we have not improved the recent current recognition complexity, our algorithm represents a new step towards a better understanding of this class. In addition, our algorithm possesses a great potential for parallel implementations.

We leave open the question of computing a “weak triangulation” of an arbitrary graph, which would help generating weakly triangulated graphs arbitrarily.

## Acknowledgments

The authors thank both Frédéric Maffray and Jerry Spinrad for open discussions on weakly triangulated graphs.

## References

1. S. Arikati and P. Rangan. An efficient algorithm for finding a two-pair, and its applications. *Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 31, 1991.
2. A. Berry. A wide-range efficient algorithm for minimal triangulation. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99)*.
3. A. Berry. *Désarticulation d'un graphe*. PhD thesis, LIRMM, Montpellier, France, 1998.

4. A. Berry, J.-P. Bordat. Minimal separators in a weakly triangulated graph. *Res. Rep. LIRMM, April 2000.*
5. A. Berry, J.-P. Bordat and O. Cogis. Generating all the minimal separators of a graph. In *Proceedings of WG'99.*
6. A. Berry, J.-P. Bordat and P. Heggernes. Computing a minimal triangulation from an arbitrary ordering. In preparation for journal submission.
7. V. Bouchitté and I. Todinca. Treewidth and minimum fill-in of weakly triangulated graphs. In *LNCS 1563, Proceedings 16th Symposium on Theoretical Aspects of Computer Science (STACS'99).* Submitted to SIAM J. Comput.
8. G.A. Dirac. On rigid circuit graphs. *Anh. Math. Sem. Univ. Hamburg,* 25:71–76, 1961.
9. D.R. Fulkerson and O.A. Gross. Incidence matrixes and interval graphs. *Pacific Journal of Math.,* 15:835–855, 1965.
10. R. Hayward. Weakly triangulated graphs. *J. Comb. Theory,* 39:200–208, 1985.
11. R. Hayward. *Two classes of perfect graphs.* PhD thesis, School of Computer Science, McGill University, 1987.
12. R. Hayward. Generating weakly triangulated graphs. *J. Graph Theory,* 21:67–70, 1996.
13. R. Hayward. Meyniel weakly triangulated graphs - 1: co-perfect orderability. *Discrete Applied Mathematics,* 73:199–210, 1997.
14. R. Hayward. Meyniel weakly triangulated graphs - 2: A theorem of Dirac. *Discrete Applied Mathematics,* 78:283–289, 1997.
15. R. Hayward, C. Hoàng, and F. Maffray. Optimizing weakly triangulated graphs. *Graphs and Combinatorics,* 5:339–349, 1989.
16. R. Hayward, J. Spinrad, and R. Sritharan. Weakly chordal graph algorithms via handles. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '2000).*
17. D. Kratsch. The structure of graphs and the design of efficient algorithms. In *Habilitation Thesis, Fakultät für Mathematik und Informatik der Friedrich-Schiller Universität Jena,* 1995.
18. C.G. Lekkerkerker and J.C. Boland. Representation of a finite graph by a set of intervals on the real line. *Fund. Math.,* 51(45–64), 1962.
19. L. Nourine and O. Raynaud, A Fast Algorithm for building Lattices. *Information Processing Letters,* 71:199–20, 1999.
20. D. Rose, R.E. Tarjan, and G. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. Comput.,* 5:146–160, 1976.
21. J. Spinrad. Finding large holes. *Information Processing Letters,* 39(4):227–229, August 1991.
22. J. Spinrad and R. Sritharan. Algorithms for weakly triangulated graphs. *Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science,* 59, 1995.
23. I. Todinca. *Aspects algorithmiques des triangulations minimales des graphes.* PhD thesis, LIP, ENS Lyon, 1999.

# Caching for Web Searching

Bala Kalyanasundaram<sup>\*,1</sup>, John Noga<sup>\*\*,2</sup>, Kirk Pruhs<sup>\*\*\*,3</sup>,  
and Gerhard Woeginger<sup>†,2</sup>

<sup>1</sup> Dept. of Computer Science  
Georgetown University  
Washington D.C. 20057 USA  
[kalyan@cs.georgetown.edu](mailto:kalyan@cs.georgetown.edu)

<sup>2</sup> Department of Mathematics  
Technical University of Graz  
Graz, Austria  
[{noga, woeginger}@opt.math.tu-graz.ac.at](mailto:{noga, woeginger}@opt.math.tu-graz.ac.at)  
<http://www.opt.math.tu-graz.ac.at/woe/index.html>

<sup>3</sup> Dept. of Computer Science  
University of Pittsburgh  
Pittsburgh, PA. 15260 USA  
[kirk@cs.pitt.edu](mailto:kirk@cs.pitt.edu)  
<http://www.cs.pitt.edu/~kirk>

**Abstract.** We study web caching when the input sequence is a depth first search traversal of some tree. There are at least two good motivations for investigating tree traversal as a search technique on the WWW: First, empirical studies of people browsing and searching the WWW have shown that user access patterns commonly are nearly depth first traversals of some tree. Secondly, (as we will show in this paper) the problem of visiting all the pages on some WWW site using anchor clicks (clicks on links) and back button clicks — by far the two most common user actions — reduces to the problem of how to best cache a tree traversal sequence (up to constant factors).

We show that for tree traversal sequences the optimal offline strategy can be computed efficiently. In the bit model, where the access time of a page is proportional to its size, we show that the online algorithm LRU is  $(1 + \frac{1}{\epsilon})$ -competitive against an adversary with *unbounded* cache as long as LRU has a cache of size at least  $(1 + \epsilon)$  times the size of the largest item in the input sequence. In the general model, where pages have arbitrary access times and sizes, we show that in order to be constant competitive, any online algorithm needs a cache large enough to store  $\Omega(\log n)$  pages; here  $n$  is the number of distinct pages in the input sequence. We provide a matching upper bound by showing that the online algorithm Landlord is constant competitive against an adversary with an unbounded cache

---

\* Supported in part by NSF Grant CCR-9734927 and by ASOSR grant F49620010011.

\*\* Supported by the START program Y43-MAT of the Austrian Ministry of Science.

\*\*\* Supported in part by NSF Grant CCR-9734927 and by ASOSR grant F49620010011.

† Supported by the START program Y43-MAT of the Austrian Ministry of Science.

if Landlord has a cache large enough to store the  $\Omega(\log n)$  largest pages. This is further theoretical evidence that Landlord is the “right” algorithm for web caching.

## 1 Introduction

### 1.1 Problem Statement and Motivation

Web caching is the temporary local storage of WWW pages by a browser for later retrieval. From the user’s point of view, the primary benefit of caching is reduced latency, as the time to access locally stored objects is minimal. We adopt the following standard general model of web caching [1,5,8,14]:

**Web Caching Problem Statement:** The browser is given an online sequence  $S$  of page requests, where each page  $p_i \in S$  has a size  $s(i)$  (say, in bytes) and an access time  $t(i)$  that is required if  $p_i$  is not cached. If the requested page  $p_i$  is not in the cache (this is called a cache miss), then the time to access  $p_i$  is  $t(i)$ . Otherwise, if  $p_i$  is in the cache (this is called a cache hit), then  $p_i$  may be accessed instantaneously. After the request of page  $p_i$ , but before the next request, the algorithm may evict/decache any arbitrary collection of pages and put  $p_i$  in its cache. At no time can the aggregate sizes of the pages in cache exceed the fixed cache size  $k$ . The objective function is to minimize the total access time.

Note that we adopt the non-forced caching model here, that is, the algorithm need not cache an accessed page. The differences between the results for forced caching and for non-forced caching models are negligible.

All of the previous work Web Caching that we are aware of assumes that the sequence  $S$  may be arbitrary. In this paper we consider the case that  $S$  is a depth first traversal of some tree  $T$  of pages. (Note that this restriction on the input allows us to obtain results that are stronger in a fundamental way.) We are motivated to consider this problem for two reasons. The first reason is that empirical studies of people browsing and searching the WWW have shown that user access patterns are commonly nearly depth first tree traversals [4,7,12]. That is, people tend to visit new pages via an anchor click (more than 50% of user actions are anchor clicks [4]), and to revisit pages using the back button (more than 40% of user actions are back button clicks [4]). No other action accounts for more than 2% of users’ actions [4]. Secondly, we show that the problem of visiting all the pages on some WWW site using anchor clicks and the back button essentially reduces to the problem of how to best cache a tree traversal sequence. This may be viewed as providing theoretical justification for tree traversal as a search technique on the WWW.

**Site Search Problem Statement:** Informally, the searcher starts at the home page  $p_h$  (say for example, [www.microsoft.com](http://www.microsoft.com)) of some WWW site (say Microsoft’s WWW site) with unknown topology. The searcher’s goal is to visit every page reachable from the home page using anchors and the back button.

More formally, an online algorithm starts at some node in an initially unknown directed graph  $G$ . Each node in  $G$  is a page  $p_i$  with size  $s(i)$  and access time  $t(i)$ . We assume that every node in  $G$  is reachable from the start page. When the online algorithm visits a node  $p_i$  it learns  $s(i)$ ,  $t(i)$ , and the names of each page  $p_j$  such that  $(p_i, p_j)$  is a directed edge in  $G$ . If  $p_i$  is not in the cache then the online algorithm must pay  $t(i)$ , otherwise the online algorithm pays nothing for this visit. After visiting  $p_i$ , the algorithm may decache any arbitrary collection of pages and put  $p_i$  in its cache. At no time may the aggregate sizes of the pages in the cache exceed  $k$ . After making its caching decision the online algorithm may make one of two moves. First, it can push  $p_i$  onto a stack  $S$ , and then move to a page  $p_j$  with the property that  $(p_i, p_j)$  is an edge in  $G$ . Second, it can pop the top page  $p_j$  off of  $S$  and return to  $p_j$ . The online algorithm must visit every page and return to the initial home page  $p_h$ . (Note that the requirement that the online algorithm return to the initial page is for convenience. Dropping this requirement will only change the competitive ratio by at most a factor of two.) The objective function is to minimize the aggregate access time of those visits where the visited page was not cached at the time of the visit.

Thus Site Search requires that the online algorithm must specify both a search strategy and a caching strategy. We show that, without loss of generality, online algorithms may restrict themselves to search strategies that traverse trees. That is, we show that the competitive ratio of every online algorithm  $A$  for the Site Search Problem is  $\Theta\left(\max_{T \in \mathcal{T}_n} \frac{A(T)}{t(T)}\right)$ , where  $\mathcal{T}_n$  is the collection of all directed rooted trees on  $n$  nodes with edges directed away from the root,  $A(T)$  is the total access time for algorithm  $A$  on the tree  $T$  assuming that it starts at the root of  $T$ , and  $t(T) = \sum_{p_i \in T} t(i)$  is the aggregate access times of the nodes in  $T$ .

Note that there are some differences between Site Search on trees and Web Caching on depth first tree traversal sequences. The online algorithm in Site Search may decide how it will traverse the tree  $T$  (this traversal need not be a depth first search traversal), while the online algorithm for Web Caching does not have this power. In Site Search, the online algorithm learns the degree of a node when it visits that node, which is not the case in Web Caching on depth first tree traversals. And most importantly, the competitive ratio for an online algorithm  $A$  for Site Search on a tree compares  $A(T)$  against the aggregate access times  $t(T)$  of the pages in  $T$ , while for Web Caching on depth first tree traversal sequences  $S$  of  $T$ , the competitive ratio compares  $A(S)$  against the optimal offline cost. We will show that the optimal offline cost may be much higher than  $t(T)$ .

There are three special cases of the caching models that have been studied previously [18]. In the *bit model* the access time is assumed to equal the size of the page. This model would be appropriate if the pages are large and the delay in the network is small. In the *cost model* the size of each page is one, while the access times are allowed to be arbitrary. This is an appropriate model if the page sizes are roughly equal. For our purposes in this paper, the cost model is really no easier for online algorithms than the general model. In the *fault model*

the access time for each page is constant, while page sizes may be arbitrary. The fault model will not interest us here since it is obvious that every online algorithm without a cache is constant competitive against an adversary with an unbounded cache in the fault model for Web Caching on depth first tree traversal sequences.

## 1.2 Our Results

Our results differ from prior work on caching in a fundamental way. In particular, we bound the size of cache required by an online algorithm in order to be constant competitive against an offline optimal algorithm that uses an *unbounded* amount of cache.

In section 2, we give the following foundational results. We show that the competitive ratio of any online algorithm for Site Search is  $\Theta\left(\max_{T \in \mathcal{T}_n} \frac{A(T)}{t(T)}\right)$ . We give a pseudo-polynomial time offline dynamic programming algorithm to compute  $\text{OPT}(S)$  when  $S$  is a depth first tree traversal. This stands in contrast to offline Web Caching for general sequences, where no pseudo-polynomial time algorithm is known [1].

In section 3, we investigate Site Search and Web Caching under the bit model. For Web Caching, we show that the online algorithm LRU is  $(1 + \frac{1}{\epsilon})$ -competitive against an adversary with *unbounded* cache as long as LRU has a cache of size at least  $(1 + \epsilon)L$ , where  $L$  is the size of the largest item in the input sequence. Note that an algorithm with unbounded cache only has to pay to access each item once; so another way to state this result is that the total access time for LRU is at most  $(1 + \frac{1}{\epsilon})$  times the aggregate access times of the pages regardless of how often these pages are accessed. Similarly, for Site Search we show that the online algorithm that uses a depth first traversal and LRU is  $(1 + \frac{1}{\epsilon})$ -competitive against an adversary with an unbounded cache as long as LRU has a cache of size at least  $(1 + \epsilon)L$ .

In section 4, we give lower bounds on the competitive ratios for Web Caching and Site Search in the cost model (obviously these also hold in the general model). We first show a lower bound of  $\Omega(\min(k, n^{1/(k+1)}))$  on the competitive ratio of any deterministic online algorithm for Web Caching. We then show a lower bound of  $\Omega\left(\max\left(\frac{1}{k}, \frac{k}{\log n}\right) n^{1/(k+1)}\right)$  on the competitive ratio of any deterministic online algorithm for Site Search. We accomplish this by showing that  $\max_{t \in \mathcal{T}_n} \frac{\text{OPT}_k(t)}{t(T)}$  is  $\Omega\left(\max\left(\frac{1}{k}, \frac{k}{\log n}\right) n^{1/(k+1)}\right)$ , where  $\text{OPT}_k(T)$  is the optimal offline cost for Site Search on  $T$ . Thus these results show that for both Web Caching and Site Search, an online algorithm needs a logarithmically sized cache to be constant competitive.

In section 5, we analyze the online algorithm Landlord (this algorithm is a generalization of LRU and is also called Greedy-Dual-Size in the literature) [3], [5], [14]. Although we will state all results in the cost model, the results hold for the general model if  $k$  is replaced by  $\frac{k}{L}$ . We show that Landlord is  $O\left(\min\left(k, \frac{\log n}{k} n^{1/(k+1)}\right)\right)$ -competitive for Web Caching on depth first tree

traversal sequences. We also show that the online algorithm that uses a depth first traversal and Landlord is  $O\left(\min\left(k, \frac{\log n}{k}\right) n^{1/(k+1)}\right)$ -competitive for Site Search. The proper way to interpret this result is that for both Site Search and for Web Caching on tree sequences, Landlord is constant competitive against an adversary with an unbounded cache if Landlord has a cache large enough to hold at least  $\log n$  pages. That is, a multiplicative increase in the number of pages only requires an additive increase in cache size to remain competitive against an adversary with infinite cache. Yet another way to interpret this result is that the number of pages that an adversary has to use to really fool Landlord is exponential in the cache size.

To date, Landlord appears to be the theoretical champion for Web Caching on arbitrary sequences [3,5,14]. Section 4 and section 5 together show that even if Landlord is not the theoretical champion for depth first tree traversal sequences, then at least it is not far away from being the champion. That is, even if one was going to design an online algorithm specifically for depth first tree traversal sequences, one could not do a whole lot better than Landlord. We take this as further theoretical evidence that Landlord is the “right” algorithm for Web Caching.

In section 6 we introduce a new algorithm Slumlord for Web Caching on depth first tree traversal sequences. Slumlord is a variation of the algorithm Landlord in that instead of raising the rent on every page in the cache, Slumlord in some sense only raises the rent on the one page that can least afford to pay (this uses the rent analogy from [14]). Thus Slumlord is a much more conservative algorithm than Landlord as it will wait longer to evict pages. We show that all of the analysis results on Landlord from section 5 also hold for Slumlord. Our purpose for introducing Slumlord is that we have some reason to suspect that in some cases, i.e. for some particular relationships between  $k$  and  $n$ , Slumlord may perform slightly better than Landlord on depth first tree traversal sequences.

Notice that LRU is what we will call an *oblivious* algorithm, in that it ignores the access times of the pages. In section 7 we consider oblivious algorithms in the cost model. We show that the online algorithm Least Frequently Evicted (LFE) is optimally competitive among oblivious online algorithms for Web Caching on tree traversal sequences. Furthermore, for Site Search we show that the online algorithm that uses a depth first traversal and LFE is strongly competitive if  $k = O(1)$ . An online algorithm  $A$  is *strongly competitive* for a problem  $\mathcal{P}$  if the competitive ratio of  $A$  is at most a constant factor worse than the competitive ratio of any other online algorithm for  $\mathcal{P}$ . This is in contrast to Site Search in the cost model with  $k = \omega(1)$ , and to Web Caching in the cost model over all ranges of  $k$ , where we show that there are no strongly competitive oblivious algorithms.

Due to space limitations, most of the proofs could not be included in this version of the paper. These proofs may be found in the full version of the paper, on the third author’s home page.

### 1.3 Related Results

We first discuss known results for offline Web Caching. It is easy to see that Web Caching in the bit model (and in the general model) is NP-hard in the ordinary sense. In [8] polynomial-time offline  $O(\log k)$ -approximation algorithms are given for the bit model and for the fault model. In [1] a polynomial-time offline  $O(1)$ -approximation is given, provided that the polynomial time algorithm is given additional  $O(L)$  cache, where  $L$  is the size of the largest page. Additionally in [1] a polynomial time offline  $O(\log(k + L))$ -approximation algorithm is derived.

Next, let us consider online Web Caching. The algorithm Greedy-Dual-Size is introduced in [3], where it is shown to be  $k$ -competitive. Greedy-Dual-Size is a generalization of the algorithm Greedy-Dual in [13] that is specific for the cost model. In [14] it is shown that Greedy-Dual-Size (this paper introduces the name Landlord for this algorithm) is  $\frac{k}{k-h+1}$ -competitive against an adversary with cache size  $h$  assuming forced caching. In [14] it is also shown that in some sense for most choices of  $k$ , the retrieval cost is either insignificant or the competitive ratio is constant. In [5] it is shown, using linear programming duality, that Greedy-Dual-Size is  $\frac{k+1}{k-h+1}$ -competitive against an adversary with cache size  $h$  assuming non-forced caching. In [8] online randomized  $O(\log^2 k)$ -competitive algorithms are given for the bit and fault models .

Previous researchers have theoretically studied the caching problem with uniform times and uniform sizes under particular input patterns. In [2] (and in several follow-up papers) the input is assumed to be a walk in a graph, and in [11] the input is assumed to be the output of a Markov chain. In [9] it is shown that if the input sequence is a depth first traversal of a tree then LRU will have  $2n - k$  cache misses, and that LRU always performs better than Most Recently Used on depth first traversal sequences.

In [6] the direct-mapped caching problem was studied with sequential access sequences. Perhaps the most closely related result to the search part of Site Search is in [10]; Recasting the results from a geometric setting to the Site Search setting, it is shown in [10] that there is an online algorithm that is constant competitive if  $k = 0$ ,  $G$  is planar, and the edge relation in  $G$  is symmetric.

## 2 Foundational Results

**Theorem 1.** *For Site Search in any model (general, cost, or bit), the competitive ratio of every deterministic online algorithm  $A$  is  $\Theta\left(\max_{T \in \mathcal{T}_n} \frac{A(T)}{t(T)}\right)$ .*

*Proof.* The competitive ratio is at most  $\max_{T \in \mathcal{T}_n} \frac{A(T)}{t(T)}$ , since the online searcher may perform a depth first traversal of the site and the offline searcher has to access every page at least once.

To see why the competitive ratio is at least  $\frac{1}{4} \max_{T \in \mathcal{T}_n} \frac{A(T)}{t(T)}$ , let  $T$  be an arbitrary directed rooted tree on  $n$  nodes with all edges directed away from the root. Let  $p_n$  be the last page in  $T$  visited by  $A$ . Create a directed graph  $G$  that includes each directed edge in  $T$  and directed edges going from  $p_n$  to every other

node in  $T$ . Then  $A$ 's actions on  $G$  are identical to  $A$ 's actions on  $T$  until  $p_n$  is visited. From  $p_n$ ,  $A$  may return directly to the root; hence,  $A(G) \geq \frac{1}{2}A(T)$ . The offline adversary may visit all of  $G$  incurring cost at most  $2t(T)$  by traversing the shortest path from the root to  $p_n$ , then visiting each remaining unvisited node in a hub and spoke pattern from  $p_n$ , and then backing up to the root.

For an instance  $T \in \mathcal{T}_n$  of Site Search, we define  $\text{OPT}_k(T)$  to be a minimum access time strategy for visiting all the nodes in  $T$  and returning to the root of  $T$  assuming that the cache size is  $k$ . We show for Site Search on trees that the optimal offline algorithm may use any depth first search that it likes. Note that this in no way implies that the optimally competitive *online* algorithm uses depth first search.

**Lemma 1.** *For every depth first traversal  $S$  of a tree  $T$ , there is an optimal Site Search strategy that uses  $S$  to traverse the tree  $T$ .*

**Lemma 2.** *For Web Caching on depth first tree traversals in the cost model, and for Site Search in the cost model,  $\text{OPT}_k(T)/t(T) \leq n^{\frac{1}{k+1}} + 1$  holds for any tree  $T$  with  $n$  nodes.*

We now give an optimal offline algorithm for Site Search on trees and for Web Caching on depth first tree traversal sequences in the general model. Let  $p_r$  be a node in  $T$  with children  $p_{c_1}, \dots, p_{c_m}$ . If  $s(r) > k$  then obviously

$$\text{OPT}_k(T_r) = t(r) + \sum_{i=1}^m [\text{OPT}_k(T_{c_i}) + t(r)]$$

So now consider the case that  $s(r) \leq k$ . We say that  $p_{c_i}$  is cheap if  $\text{OPT}_{k-s(r)}(T_{c_i}) - \text{OPT}_k(T_{c_i}) < t(r)$ , and otherwise we say that  $p_{c_i}$  is expensive. It is easy to see that one should cache  $p_r$  before visiting a cheap child  $p_{c_i}$  since the time savings from having additional  $s(r)$  cache is less than the access time for  $p_r$ . Similarly, one should not cache  $p_r$  before visiting an expensive child  $p_{c_i}$  since one can reap a time savings of more than  $t(p_r)$  by having additional  $s(r)$  cache during the traversal of  $T_{c_i}$ . Hence,

$$\text{OPT}_k(T_r) = t(r) + \sum_{\text{cheap } p_{c_i}} \text{OPT}_{k-s(r)}(T_{c_i}) + \sum_{\text{expensive } p_{c_i}} [\text{OPT}_k(T_{c_i}) + t(r)]$$

The obvious dynamic programming implementation of this recurrence runs in time  $O(kn)$ . Summarizing, this dynamic program yields a pseudo-polynomial time algorithm for Web Caching of tree traversal input sequences in the bit model and in the general model, and a polynomial time algorithm for the cost model.

### 3 Bit Model

The algorithm Least Recently Used (LRU) evicts the least recently used items until there is room to fit the most recently requested item in the cache. We show that in the bit model the online algorithm LRU is  $(1 + \frac{1}{\epsilon})$ -competitive against an adversary with unbounded cache as long as LRU has a cache of size at least  $(1 + \epsilon)L$ . Recall  $L$  is the size of the largest item in the input sequence.

**Theorem 2.** Suppose  $0 < \epsilon \leq 1$  and that that LRU is equipped with a cache of size  $k \geq (1 + \epsilon)L$ . Then for Web Caching in the bit model where the input sequence  $S$  is a depth first traversal of some tree  $T$ , the algorithm LRU guarantees that  $\frac{LRU(S)}{t(T)} \leq 1 + \frac{1}{\epsilon}$ .

*Proof.* We split the cost of LRU into the cost incurred while moving downwards (from a parent down to a child) and the cost incurred while moving upwards (from a child up to its parent). We show by an amortization argument, that the total cost for upward moves is at most  $\frac{t(T)}{\epsilon}$ . There is an account  $ACC_i$  associated with each page  $p_i$ , and there is an account  $ACC(LRU)$  for LRU. Initially,  $ACC_i = \frac{t(i)}{\epsilon}$  for each page  $p_i$  and  $ACC(LRU) = 0$ . When a page  $p_i$  is requested in a downward move, all accounts remain unchanged. When a node  $p_i$  is requested in an upward move and  $p_i$  is not cached, then  $t(i)$  is deducted from  $ACC(LRU)$ . If the request sequence is next going to visit another child of  $p_i$ , then all the funds in  $ACC(LRU)$  are moved to  $ACC_i$ , and LRU enters this subtree with an empty account. Otherwise, if the request sequence returns to  $p_i$ 's parent, then all the funds in  $ACC_i$  are transferred to  $ACC(LRU)$ .

Our first goal is to show that during an upward move from a node  $p_i$  towards its parent,  $ACC(LRU) \geq \min(\frac{t(T_i)}{\epsilon}, L)$  always holds. The proof is done by induction. The base case is if  $p_i$  is a leaf. In this case the account of  $p_i$  with value  $\frac{t(i)}{\epsilon}$  has just been transferred to LRU, and thus  $ACC(LRU) \geq \frac{t(i)}{\epsilon} = \frac{t(T_i)}{\epsilon}$  holds. Next assume that the claim holds for each of the children  $p_{c_1}, \dots, p_{c_m}$  of  $p_i$ . We break the proof into two cases: (Case 1) First, assume that for all  $j$ ,  $1 \leq j \leq m$ ,  $t(T_{c_j}) \leq \epsilon L$  holds. Then every  $T_{c_j}$  can be traversed without evicting  $p_i$ , and  $p_i$  will be kept cached throughout the traversal of  $T_i$ . Since no charges are deducted from the searchers account at  $p_i$ , the inductive claim yields that  $ACC(LRU) \geq \frac{t(i)}{\epsilon} + \sum_{j=1}^m \frac{t(T_{c_j})}{\epsilon} = \frac{t(T_i)}{\epsilon}$  holds at the moment when LRU leaves  $p_i$  upwards to its parent. (Case 2) Now assume that there exists a  $j$ ,  $1 \leq j \leq m$ , with  $t(T_{c_j}) > \epsilon L$  and consider the moment in time when the searcher returns from  $p_{c_j}$  up to  $p_i$ . At this moment,  $p_i$  need not be in the cache. By induction, the value of  $ACC(LRU)$  is at least  $L = \min(\frac{t(T_{c_j})}{\epsilon}, L)$ . Hence, after (possibly) paying the charge for visiting  $p_i$ ,  $ACC(LRU) \geq L - t(i)$  holds. If the request sequence now returns to  $p_i$ 's parent, then  $ACC(LRU) \geq (L - t(i)) + \frac{t(i)}{\epsilon} \geq L$  (where the second term is the original amount in  $ACC_i$ ). Otherwise, if the request sequence moves on to the next child of  $p_i$ , then  $ACC(LRU) \geq L - t(i)$  will be added to  $ACC_i$  and so  $ACC_i \geq (L - t(i)) + \frac{t(i)}{\epsilon} \geq L$  and this amount will eventually be transferred to  $ACC(LRU)$  before it moves up to  $p_i$ 's parent.

Next, we argue that  $\text{ACC}(\text{LRU})$  is never negative, and that therefore LRU can always pay for the revisits. Consider visiting a parent  $p_i$  from a child  $p_c$ . If  $t(T_c) \leq \epsilon L$  then  $p_i$  is still cached at this moment and no charge is taken. Otherwise, if  $t(T_c) > \epsilon L$  then  $\text{ACC}(\text{LRU}) \geq L$  and LRU can afford the charge since  $t(i) \leq L$  by the definition of  $L$ . Summarizing, the account of LRU stays non-negative throughout the traversal of the tree. Since the total amount of funds available in the beginning is  $\frac{t(T)}{\epsilon}$  and since LRU is able to finance all its upward moves from these funds, the total incurred cost indeed is at most  $\frac{t(T)}{\epsilon}$ . Since the total cost of LRU for downward moves equals  $t(T)$ , the proof of the theorem is complete.

This corollary is an immediate consequence of theorem 1 and theorem 2.

**Corollary 1.** *For Site Search in the bit model, the algorithm that uses LRU and a depth first traversal guarantees that  $\frac{\text{LRU}(S)}{t(T)} \leq 1 + \frac{1}{\epsilon}$ .*

It is easy to see that the above bounds are tight for  $\epsilon = 1$  by considering trees where each internal node has one child, and all pages have access time  $L$ . Note that in Site Search that this is not merely an artifice of our requirement that the searcher return to the root as you could always enforce this condition by adding a second leaf-child of the root.

## 4 Lower Bounds in the General Model

We show that in the cost model (and hence also in the general model), every online algorithm for Web Caching and every online algorithm for Site Search requires a cache of size  $\Omega(\log n)$  in order to be constant competitive.

**Theorem 3.** *For Web Caching in the cost model, any deterministic online algorithm  $A$  fulfills the following statements.*

- (i) *Let  $k$  and  $n$  be integers such that  $k + 1 \leq \lg n$ . Then there exists a tree  $T$  with  $\Theta(n)$  nodes on which  $A$  is  $\Omega(\min(k + 1, n^{1/(k+1)}))$ -competitive.*
- (ii) *Let  $k$  and  $n$  be integers such that  $k + 1 \leq \lg n$ . Then there exists a tree  $T$  with  $\Theta(n)$  nodes such that  $A(T)/t(T) \geq \frac{1}{4}n^{\frac{1}{k+1}}$ .*

*Proof.* The adversary constructs a tree  $T$  with  $k + 2$  levels numbered  $0, 1, \dots, k + 1$ . Level 0 only contains the root of  $T$ , level 1 contains all the children of the root, and so on. Every page at level  $\ell$  has access time  $x^{k+1-\ell}$ , where  $x = \frac{1}{2}n^{1/(k+1)}$ . Note that  $x \geq 1$  since  $k + 1 \leq \lg n$ . Hence, every node has access time at least one. The exact shape of  $T$  is determined by the adversary in dependence on the behavior of the online algorithm  $A$ . The adversary follows a simple *Hit-Where-It-Hurts* strategy. Let  $p$  be the last requested page, and let  $\ell$  be the level that contains  $p$ .

**Expand:** If  $\ell < k + 1$ , then the adversary creates a path of  $k + 1 - \ell$  new pages at levels  $\ell + 1, \dots, k + 1$  that are descendants of page  $p$ . The pages on this path are then requested one by one.

**Hit:** Otherwise,  $\ell = k + 1$  holds. The adversary requests the ancestors of  $p$  until it reaches a page that is currently not cached by the online algorithm.

The adversary alternates between expansions and hits until it has created  $n$  nodes (if this happens in the middle of an expansion or hit, this move is still completed and then the process stops). Clearly, the thus created tree  $T$  has  $\Theta(n)$  nodes. By  $n_\ell$ ,  $0 \leq \ell \leq k + 1$ , we denote the total number of nodes at the  $\ell$ -th level of tree  $T$ . Note that  $n_0 = 1$ .

Now let  $p$  be a page at level  $\ell \leq k$  with  $m$  children. Since all leaves of  $T$  are at level  $k + 1$ ,  $m \geq 1$  holds. When the online algorithm pays for accessing  $p$  then either the adversary is expanding the tree (and  $p$  is created) or the adversary is hitting (and the request sequence returns from one of the  $m$  children). When the request sequence returns from one of the first  $m - 1$  children, the adversary just has done a hit. The online algorithm pays for accessing  $p$ , and then the next child is created in the following expansion. When the request sequence returns from the last child, it immediately moves on to the parent of  $p$  and we are in the middle of some hit. Altogether, for accessing page  $p$ , the algorithm  $A$  pays  $m$  times the size of  $p$ , and for all the accesses to all the pages in level  $\ell$ , it pays the total number  $n_{\ell+1}$  of their children times their access time  $x^{k+1-\ell}$ . For the pages in level  $k + 1$ ,  $A$  altogether pays  $n_{k+1}$  times access time 1. Summarizing, this yields

$$A(T) = n_{k+1} + \sum_{\ell=0}^k n_{\ell+1} x^{k+1-\ell} \geq xt(T) - x^{k+2} = x(t(T) - n/2^{k+1}) \geq \frac{x}{2}t(T). \quad (1)$$

In the last inequality, we used that  $t(T) \geq n$ . This inequality holds since every node has access time at least one.

One possible offline strategy always keeps all the predecessors of the currently requested page in cache, with the exception of the pages at some fixed level  $\lambda$  with  $0 \leq \lambda \leq k$ . Since  $T$  has only  $k + 2$  levels and since there is no need to cache the pages at level  $k + 1$ , this strategy can always be carried out with a cache of size  $k$ . This offline strategy has to pay for accessing a page (a) if the page is requested for the first time, or (b) if the page is at level  $\lambda$  and if the request sequence moves from a page at level  $\lambda + 1$  up to level  $\lambda$ . The total cost for (a) is  $t(T)$ , and the total cost for (b) is  $n_{\lambda+1}x^{k+1-\lambda}$ . Hence,  $\text{OPT}(T) \leq t(T) + \min_{\lambda=0}^k \{n_{\lambda+1}x^{k+1-\lambda}\}$ , and a simple averaging argument yields

$$\text{OPT}(T) \leq t(T) + \frac{1}{k+1} \sum_{\ell=0}^k n_{\ell+1} x^{k+1-\ell} \leq t(T) + \frac{1}{k+1} x \cdot t(T). \quad (2)$$

By combining (1) and (2), we conclude that the competitive ratio of  $A$  is at least

$$A(T)/\text{OPT}(T) \geq \frac{(k+1)x \cdot t(T)}{2(k+1+x)t(T)} = \frac{(k+1)x}{2(k+1+x)} \quad (3)$$

Now let us prove statement (i) of the theorem. If  $k+1 \leq n^{1/(k+1)}$ , then  $k+1 \leq 2x$  and we derive from (3) that

$$A(T)/\text{OPT}(T) \geq \frac{(k+1)x}{2(k+1+x)} \geq \frac{(k+1)x}{6x} = \frac{1}{6}(k+1).$$

If on the other hand  $k+1 \geq n^{1/(k+1)}$  holds, then  $k+1 \geq x$  and we derive in a similar way that

$$A(T)/\text{OPT}(T) \geq \frac{(k+1)x}{2(k+1+x)} \geq \frac{(k+1)x}{4(k+1)} = \frac{1}{8}n^{1/(k+1)}.$$

This proves (i). Finally, statement (ii) follows from the inequality in (1) and from  $x = \frac{1}{2}n^{1/(k+1)}$ . With this, the proof of the theorem is complete.

Now we turn to the Site Search problem. We know from theorem 1 that without loss of generality (and up to constant factors) we only need to consider online algorithms  $A$  that traverse some subtree  $T$  of  $G$ . However, we do not necessarily know that  $A$  performs a depth first traversal of  $T$ . To get around this difficulty, we consider the following Modified Site Search problem. It is easy to see that a lower bound on the competitive ratio for any online algorithm for Modified Site Search also yields a lower bound on the competitive ratio for any online algorithm for the original Site Search problem.

**Modified Site Search Problem:** The online algorithm is told that the topology of  $G$  consists of a directed tree  $T$ , rooted at the initial page with the edges directed away from the initial page, and edges directed from a secret page  $p_s$  to every other page. The online algorithm is told  $T$  a priori, but is not told the identity of the secret node  $p_s$  (and actually, the adversary will make  $p_s$  the last node that the online algorithm visits). The goal of the online algorithm is still to visit all the nodes and to return to the initial page.

Recall that  $\text{OPT}_k(T)$  is the minimum access time strategy, with cache size  $k$ , for visiting all the nodes in  $T$  and returning to the root of  $T$ . Also recall that by lemma 1 we may assume that  $\text{OPT}_k(T)$  uses a depth first search. Note that  $\text{OPT}_k(T)$  can be computed by the online algorithm before it begins its traversal. The competitive ratio of any online algorithm for Modified Site Search is then  $\Omega\left(\max_T \frac{\text{OPT}_k(T)}{t(T)}\right)$ . We show that  $\max_T \frac{\text{OPT}_k(T)}{t(T)}$  is at least  $\Omega\left(\max\left(\frac{1}{k}, \frac{k}{\log n}\right) n^{\frac{1}{k+1}}\right)$ .

**Theorem 4.** *For Modified Site Search, the competitive ratio of every deterministic online algorithm  $A$  is at least  $\Omega\left(\max\left(\frac{1}{k}, \frac{k}{\log n}\right) n^{\frac{1}{k+1}}\right)$ .*

**Corollary 2.** *For Site Search in the cost model, the competitive ratio of every deterministic online algorithm  $A$  is at least  $\Omega\left(\max\left(\frac{1}{k}, \frac{k}{\log n}\right) n^{\frac{1}{k+1}}\right)$ .*

## 5 Analysis of Landlord

We show that for Web Caching on depth first tree traversal sequences and for Site Search, Landlord is constant competitive against an adversary with unbounded cache if Landlord has a cache large enough to hold at least  $\log n$  pages.

**Landlord Description:** [5] The algorithm maintains a non-negative credit  $c(i)$  for each page  $p_i$  in the cache. Given a request for  $p_i$ , if  $p_i$  is in the cache the algorithm resets  $c(i)$  to  $t(i)$ . Otherwise, the algorithm sets  $c(i) = t(i)$  and “pretends”  $p_i$  is in the cache. Then it repeats the following eviction step while the total size of the items in the cache exceeds  $k$ .

*Eviction step:* Let  $p_m$  be a page in the cache that minimizes the ratio  $\frac{c(m)}{s(m)}$  and let  $\delta = \frac{c(m)}{s(m)}$ . For every  $p_i$  in the cache, the algorithm decreases  $c(i)$  by  $\delta s(i)$ , and then evicts  $p_m$ .

**Proposition 1.** [5,14] For Web Caching in the general model, Landlord is  $\frac{k+1}{k-h+1}$ -competitive against an adversary with a cache of size  $h \leq k$ .

**Theorem 5.** For Site Search in the cost model, the online algorithm that uses depth first search for traversing and Landlord for caching is

- (i)  $O\left(kn^{\frac{1}{k+1}}\right)$ -competitive if  $k \leq \sqrt{\log n}$ ,
- (ii)  $O\left(\frac{\log n}{k}n^{\frac{1}{k+1}}\right)$ -competitive if  $\sqrt{\log n} < k < \frac{1}{2}\log n$ ,
- (iii)  $O(1)$ -competitive if  $k \geq \frac{1}{2}\log n$ .

**Theorem 6.** For Web Caching in the cost model, Landlord is  $O\left(\min\left(k, \frac{\log n}{k}n^{\frac{1}{k+1}}\right)\right)$ -competitive for  $k \leq \frac{1}{2}\log n$  caches, and  $O(1)$ -competitive for  $k \geq \frac{1}{2}\log n$  caches.

## 6 Slumlord

**Slumlord Description:** The algorithm is identical to Landlord, except for two changes. Firstly, a page  $p$  is decached if the user hits the back button from  $p$ . Secondly the eviction step is different.

*Eviction step:* Let  $p_i$  be currently requested page and let  $p_j$  be the cache page (other than  $p_i$ ) that was most recently requested. Let  $\delta = \min\left(\frac{c(i)}{s(i)}, \frac{c(j)}{s(j)}\right)$ . The algorithm decreases  $c(i)$  by  $\delta s(i)$  and decreases  $c(j)$  by  $\delta s(j)$ . The algorithm then evicts one of  $p_i$  and  $p_j$  with zero credits.

We call the algorithm Slumlord for the following reason. In [14] the decrementing of the credits was thought of as being analogous to raising rents. In the worst case trees the nodes lower in the trees have lower access times. So Slumlord only raises the rent on those nodes lowest in tree, which are also the ones that can least afford to pay.

**Theorem 7.** For Web Caching, Slumlord is  $\frac{k+1}{k-h+1}$ -competitive against an adversary with a cache of size  $h \leq k$ .

**Theorem 8.** For Site Search in the cost model, the online algorithm that uses a depth first traversal and Slumlord for caching is  $O\left(\min\left(k, \frac{\log n}{k}\right) n^{\frac{1}{k+1}}\right)$ -competitive.

**Theorem 9.** For Web Caching in the cost model, Slumlord is  $O\left(\min\left(k, \frac{\log n}{k} n^{\frac{1}{k+1}}\right)\right)$ -competitive.

## 7 Oblivious Algorithms for the Cost Model

We show that Least Frequently Evicted (LFE) is essentially the best oblivious algorithm and that the algorithm for Site Search that uses depth first and LFE is strongly competitive if  $k = O(1)$ . Recall that an oblivious algorithm is one that ignores access times.

**LFE Description:** An eviction count  $e(p_i)$  is maintained for each page  $p_i$ . Initially each  $e(p_i) = 0$ . Assume that the page  $p_r$  has just been requested at time  $u$ . If this is not the first time that  $p_r$  was requested, the page  $p_c$  requested at time  $u-1$  is decached if  $p_c$  is in the cache (note that  $p_c$  is a child of  $p_r$ ). As a consequence of this, LFE maintains the invariant that all the pages that are in the cache are on the path from the root to the last requested page. If the cache is not full just before  $p_r$  was requested then  $p_r$  is added to the cache. If the cache was full before  $p_r$  was requested, then LFE pretends that  $p_r$  is in the cache and selects a  $p_i$  in the cache that minimizes  $e(p_i)$ ; in case of a tie,  $p_i$  is selected to the the page closest to the root. Note that it may be the case that  $i = r$ . The selected page  $p_i$  is then evicted and  $e(p_i)$  is incremented.

For fixed  $n$  and  $k$ , let  $\gamma = \gamma(n, k)$  be the smallest integer that satisfies  $\gamma \binom{\gamma+k}{\gamma} = \gamma \binom{\gamma+k}{k} \geq n$ . Observe that  $(k+1) \left(\frac{\gamma}{k+1}\right)^{k+1} \leq \gamma \binom{\gamma+k}{\gamma} \leq (k+1) \left(\frac{e(\gamma+k)}{k+1}\right)^{k+1}$ . For  $k \leq \frac{1}{4} \log n$ , we have  $\gamma = \Theta(kn^{\frac{1}{k+1}})$ . We will call a page *fat* if it has at least  $\gamma$  children, and otherwise we call the page *skinny*. Define  $a(k, \ell)$  as the minimum over all trees of the number of distinct fat pages that must be requested before LFE, with a cache of size  $k$ , causes the eviction count of some page to reach  $\gamma + \ell$ . We now state some preliminary lemmas that are necessary for the analysis of LFE.

**Lemma 3.** If  $1 \leq \ell \leq \gamma + 1$  then  $a(k, \ell) \geq \binom{k+\ell-1}{\ell-1}$ .

**Theorem 10.** For Web Caching in the cost model, LFE is  $(2\gamma+2)$ -competitive, and hence,  $\Theta(kn^{\frac{1}{k+1}})$ -competitive.

**Corollary 3.** For Site Search in the cost model, the algorithm that uses LFE and depth first search is  $O(kn^{\frac{1}{k+1}})$ -competitive.

We now show that every oblivious online algorithm for Web Caching has competitive  $\Omega(kn^{\frac{1}{k+1}})$ -competitive. Since any page could have nonzero access time while all other pages have zero access time, an  $\gamma$ -competitive oblivious algorithm cannot miss any page more than  $\gamma$  times.

**Theorem 11.** *For Web Caching in the cost model, every deterministic oblivious online algorithm is  $\Theta(kn^{\frac{1}{k+1}})$ -competitive.*

## References

1. S. Albers, S. Arora, and S. Khanna, “Page replacement for general caching problems”, ACM/SIAM Symposium on Discrete Algorithms, 31–40, 1999.
2. A. Borodin, S. Irani, P. Raghavan, and B. Schieber, “Competitive paging with locality of reference”, *Journal of Computer and System Sciences* **50**, 244–258, 1995.
3. P. Cao and S. Irani, “Cost-aware WWW proxy caching algorithms”, USENIX Symposium on Internet Technologies and Systems, 193–206, 1997.
4. L. Catledge and J. Pitkow, “Characterizing browsing strategies in the world wide web”, *Computer Networks and ISDN Systems* **27**, 1065–1073, 1995.
5. E. Cohen, and H. Kaplan, “Caching documents with variable sizes and fetching costs: an LP based approach”, ACM/SIAM Symposium on Discrete Algorithms, S879–S880, 1999.
6. R.E. Ladner, J.D. Fix, and A. LaMarca, “Cache performance analysis of traversal and random accesses”, ACM/SIAM Symposium on Discrete Algorithms, 613–622, 1999.
7. B. Huberman, P. Pirolli, J. Pitkow, and R. Lukose, “Strong regularities in world wide web surfing”, *Science* **280**, 95–97, 1998.
8. S. Irani, “Page replacement with multi-size pages and applications to web caching”, ACM Symposium on Theory of Computing, 701–710, 1997.
9. B. Jiang, “DFS-traversing graphs in a paging environment, LRU or MRU”, *Information Processing Letters* **40**, 193–196, 1991.
10. B. Kalyanasundaram and K. Pruhs, “Constructing competitive tours from local information”, *Theoretical Computer Science* **130**, 125–138, 1994.
11. A. Karlin, S. Phillips, and P. Raghavan, ”Markov paging”, IEEE Symposium on Foundations of Computer Science, 208–217, 1992.
12. L. Tauscher and S. Greenberg, “How people revisit web pages: empirical findings and implications for the design of history systems”, *International Journal of Human-Computer Studies* **47**, 97–137, 1997.
13. N. Young, “The k-server dual and loose competitiveness”, *Algorithmica* **11**, 525–541, 1994.
14. N. Young, “On-line file caching”, ACM/SIAM Symposium on Discrete Algorithms, 82–86, 1998.

# On-Line Scheduling with Precedence Constraints

Yossi Azar<sup>1,\*</sup> and Leah Epstein<sup>1</sup>

Dept. of Computer Science, Tel-Aviv University. {azar,lea}@math.tau.ac.il

**Abstract.** We consider the on-line problem of scheduling jobs with precedence constraints on  $m$  machines. We concentrate in two models, the model of uniformly related machines and the model of restricted assignment. For the related machines model, we show a lower bound of  $\Omega(\sqrt{m})$  for deterministic and randomized on-line algorithms, with or without preemptions even for jobs of known durations. This matches the deterministic upper bound of  $O(\sqrt{m})$  given by Jaffe for task systems. The lower bound should be contrasted with the known bounds for jobs without precedence constraints. Specifically, without precedence constraints, if we allow preemptions then the competitive ratio becomes  $\Theta(\log m)$ , and if the durations of the jobs are known then there are  $O(1)$  competitive (preemptive and non-preemptive) algorithms.

We also consider the restricted assignment model. For the model with consistent precedence constraints, we give a (randomized) lower bound of  $\Omega(\log m)$  with or without preemptions. We show that the (deterministic) greedy algorithm (no preemptions used), is *optimal* for this model i.e.  $O(\log m)$  competitive. However, for general precedence constraints, we show a lower bound of  $m$  which is easily matched by a greedy algorithm.

## 1 Introduction

We consider the on-line problem of scheduling a sequence of jobs with precedence constraints on  $m$  parallel machines. A job can be scheduled after all its predecessors are completed. In the simplest model, the identical machines model, each job  $j$  has a running time  $w_j$ , and has to be scheduled on a machine for this period of time.

In the related machines model each machine  $i$  has a speed  $v_i$ . Each job may be processed on any machine and the time to process a job  $j$  with a running time  $w_j$  on  $i$  would be  $w_j/v_i$ . In the restricted assignment model all machines have identical speed, but each job may be assigned only to a subset of the machines. For a job  $j$ , we denote by  $M(j) \subseteq \{1, \dots, m\}$  ( $M(j) \neq \emptyset$ ) the subset of machines on which it may be scheduled and by  $w_j$  its running time on a machine is  $M(j)$ . The unrelated machines model is a generalization of all previous models. In this model, each job  $j$  has a vector of  $m$  components, where for each  $i$  component  $i$  gives its running time on machine  $i$ .

---

\* Research supported in part by the Israel Science Foundation and by the United States-Israel Binational Science Foundation (BSF).

We may or may not allow preemptions. If no preemptions are allowed, once a job is scheduled on a machine, it must be processed on this machine continuously until it is completed. Otherwise, if we allow preemption, a job may be stopped, and resumed later on some (maybe different) machine.

The precedence constraints between jobs can be viewed as a directed graph  $G$ . The vertices of  $G$  are the jobs. An edge  $(j_1, j_2)$  occurs when  $j_1$  is a predecessor of  $j_2$ , i.e.  $j_2$  may start its process only after  $j_1$  is completed. For restricted assignment the precedence constraints are called consistent if for every edge  $(j_1, j_2)$  we have  $M(j_2) \subseteq M(j_1)$ . The motivation for consistent precedence constraints comes from the fact that if a job  $j_1$  requires some expertise which are known only to some machines and  $j_1$  is a predecessor for another job  $j_2$ , then  $j_2$  should require at least the same expertise and hence can be processed only on subset of machines that  $j_1$  can be processed on.

We discuss an on-line environment in which a job becomes known as soon as all its predecessors are completed (there are no realize times). The goal is to minimize the makespan which is the time that the last job is finished. We consider two variations of the on-line model. In the known duration case, the durations of a job is known upon its arrival, and in the unknown duration case, the duration of a job becomes known only when it departs. Our lower bounds hold even for the known duration case (and hence also for the unknown duration case) while the algorithms do not use the informations on the durations and therefore are valid for both cases. For a survey on on-line scheduling we refer the reader to [8].

We measure the algorithms in terms of the competitive ratio. We compare the cost (makespan) of the on-line algorithm (denoted by  $C_{on}$ ) to the cost of the optimal off-line algorithm that knows the sequence in advance (denoted by  $C_{opt}$ ). The off-line algorithm knows all jobs and their properties (running time, precedence constraints and assignment restrictions) in advance. Note that the on-line algorithm is familiar with all properties of a job as soon as the job arrives (except for the running time, in the case of unknown durations), but a job arrives only after all its predecessors are completed. A deterministic algorithm is  $r$  competitive (has competitive ratio  $r$ ) if  $C_{on} \leq rC_{opt}$ . If the algorithm is randomized, we use the expectation of the on-line cost instead of the cost and the competitive ratio is  $r$  if  $E(C_{on}) \leq rC_{opt}$ .

**Our results.** For related machines we give a deterministic and randomized lower bound of  $\Omega(\sqrt{m})$  on the competitive ratio of any on-line algorithm for jobs with precedence constraints. This matches the upper bound of Jaffe [7] who gave an approximation algorithm which can be implemented in an on-line environment. In fact, Davis and Jaffe [3] already gave a lower bound of  $\Omega(\sqrt{m})$  for the case with no precedence constraints which obviously holds for the case of precedence constraints. However, their lower bound is valid only for unknown durations and no preemption. If we allow preemption then Shmoys, Wein and Williamson [9] showed an upper bound of  $O(\log m)$  for the case of no precedence constraints. Moreover, if the durations are known then in principle one can get 1 competitive algorithm for both preemptive and non-preemptive cases. This

follows from the fact that if there are no precedence constraints then it implies that all the jobs are known in advance and the problem becomes an off-line one. This should be contrasted with our result that implies that with precedence constraints one cannot get a better bound than  $\Theta(\sqrt{m})$  even if the duration are known and the algorithms are preemptive. Moreover, our lower bound holds for randomized preemptive online algorithm versus deterministic non-preemptive adversary.

For the restricted assignment model we consider the greedy algorithm which is an adaptation of the LIST algorithm of Graham [5,6]. This algorithm achieves a competitive ratio of  $2 - 1/m$  for scheduling jobs with precedence constraints on identical machines. Epstein [4] shows that LIST is optimal for scheduling jobs with precedence constraints on identical machines, even if preemptions are allowed. Azar et al [1] showed that for the case of no precedence constraints the greedy algorithm for scheduling jobs one by one in the restricted assignment model achieves a competitive ratio of  $O(\log m)$ . We show that if we allow consistent precedence constraints then the competitive ratio of the algorithm is still  $O(\log m)$ . We show that the algorithm is optimal in this case by giving a lower bound of  $\Omega(\log m)$  on the competitive ratio of any deterministic or randomized algorithm for scheduling jobs with restricted assignment and consistent precedence constraints. We note that our lower bound does not follow from the lower bound of [1] since here we do not insist on scheduling a job immediately upon its arrival. Our lower bound holds even for the known duration case and the upper bound does not use the durations. Moreover, the lower bound holds even for randomized preemptive algorithm versus deterministic non-preemptive adversary while the upper bound holds for non-preemptive algorithm versus preemptive adversary. Again, the precedence constraints are crucial for proving the lower bounds with known durations, since, otherwise, it becomes an off-line problems since all jobs are given at the beginning, and durations are known in advance.

For general precedence constraints we show a lower bound of  $m$  for any online algorithms ( $\Omega(m)$  for randomized algorithms). This bound is easily matched by the greedy algorithm which is  $m$  competitive. This implies that the unrelated machines case is not of an interest since it is  $m$  competitive.

**The Greedy algorithm.** We adapt the Greedy algorithm "List", given by Graham [5] for identical machines, to the case of restricted assignment as follows. Each time that a machine  $i$  becomes idle, assign to it a job  $j$  (if exists) such that  $i \in M(j)$  and  $j$  has not been scheduled yet. Each time that a new job  $j$  arrives, assign it to an idle machine  $i \in M(j)$  if exists. Note that Greedy is deterministic and does not use preemptions.

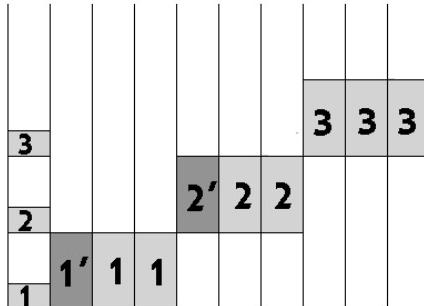
**Randomized algorithms.** To prove lower bounds on the competitive ratio of randomized algorithms we use an adaptation of Yao's theorem for on-line algorithms. It states that if there exists a probability distribution on the input sequences for a given problem such that  $E(C_{on}/C_{opt}) \geq c$  for all deterministic on-line algorithms, then  $c$  is a lower bound on the competitive ratio of all randomized

algorithms for the problem. (see [2]). We will use only sequences for which  $C_{opt}$  is constant and thus in our case  $E(C_{on}/C_{opt}) = E(C_{on})/C_{opt}$ .

## 2 Scheduling on Related Machines

**Theorem 1.** Any on-line algorithm for scheduling on related machines has the competitive ratio of at least  $\Omega(\sqrt{m})$ . This is true even for randomized preemptive algorithms versus deterministic non-preemptive adversary.

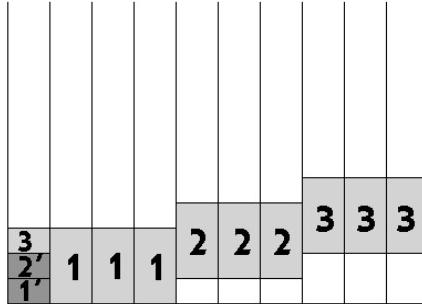
*Proof.* We start by considering deterministic algorithms. We assume without loss of generality that  $m - 1$  is a square,  $\sqrt{m - 1} = r$ . The set of machines consists of one fast machine of speed  $r = \sqrt{m - 1}$  (machine  $m$ ) and  $m - 1$  slow machines of speed 1 (machines  $1, \dots, m - 1$ ). There are  $r$  phases of  $r + 1$  unit jobs each in the sequence. The sequence begins with  $r + 1$  independent unit jobs (phase 1). Next we define phase  $i$ ,  $2 \leq i \leq r$ , the phase contains  $r + 1$  units jobs. Let  $b_{i-1}$  be the job in phase  $i - 1$  that finishes last by the on-line algorithm, then all jobs of phase  $i$  depend on  $b_{i-1}$ .



**Fig. 1.** A possible on-line assignment in the proof of Theorem 1

The on-line algorithm, by the definition of  $b_i$ , can start scheduling phase  $i+1$  only after all jobs of phase  $i$  are completed. Since each phase consists of  $r+1$  jobs, it is possible to use at most  $r+1$  machines at each time. The  $r+1$  fastest machines can process at most  $2r$  unit jobs in one unit of time, and since the total running time of all jobs in one phase is  $r+1$ , each phase takes at least  $(r+1)/(2r) > 1/2$  time units. Thus the total time to process all the sequence is at least  $r((r+1)/(2r)) = (r+1)/2 = \Omega(\sqrt{m})$  (see Figure 1).

The optimal off-line algorithm assigns each  $b_i$  to the fast machine at time  $(i-1)/r$ , and thus the jobs of phase  $i+1$  may be assigned at time  $i/r$  to machines  $ir+1, \dots, (i+1)r$ . The jobs of phase  $r$  would finish at time  $(r-1)/r + 1 < 2$  on the slow machines. The fast machine would finish at time 1 and thus  $C_{opt} < 2$  (see Figure 2). The competitive ratio is  $\Omega(\sqrt{m})$ .



**Fig. 2.** A possible off-line assignment in the proof of Theorem  $\square$

To extend the proof for randomized algorithms,  $b_i$  is chosen uniformly at random among all jobs of phase  $i$ . Clearly the optimal schedule remains the same. Next we evaluate the expected on-line schedule. The probability that the time starting from the arrival of phase  $i$ , till  $b_i$  is completed is at least  $T$  would be  $(r+1-k)/(r+1)$  where  $k$  is the maximum number of jobs that it is possible to complete in time  $T$ . For  $T = (r+1)/(4r)$ , it is possible to complete at most  $\lfloor(r+1)/2\rfloor$  jobs and thus the expectation of the time that passes from the arrival of  $b_i$  and till it is completed is at least  $(r+1)/(8r) > 1/8$ , and thus  $E(C_{on}) = \Omega(\sqrt{m})$  and again the competitive ratio is  $\Omega(\sqrt{m})$  as well.

### 3 Restricted Assignment with Consistent Precedence Constraints

In this section we consider consistent precedence constraints for the restricted assignment model. Recall that precedence constraints are called consistent if for every  $j_1$  which is a predecessor of  $j_2$  we have  $M(j_2) \subseteq M(j_1)$ .

**Theorem 2.** *Any on-line scheduling algorithm for the restricted assignment model with consistent precedence constraints has a competitive ratio of at least  $\Omega(\log m)$ . This is true even for randomized preemptive algorithms versus deterministic non-preemptive adversary.*

*Proof.* We assume without loss of generality that  $m$  is a power of 2,  $m = 2^k$ . It is easy to extend the proof for general  $m$ . The sequence consists of  $mN$  jobs where  $N > 2 \log_2 m = 2k$ , the jobs belong to  $k+1$  phases, where for  $1 \leq i \leq k$  phase  $i$  contains  $m(N+2-i)/2^i$  unit jobs, and phase  $k+1$  contains  $N - \log_2 m$  unit jobs. The jobs of phase  $i$  are restricted to machines  $\{1, \dots, 2^{k-i+1}\}$ . We define the dependencies according to the behavior of the on-line algorithm. Let  $b_1$  the job that finishes last in phase 1, then all jobs of phase 2 depend on  $b_1$ .

For  $i = 2, \dots, k$ , let  $b_i$  the job that finishes last from phase  $i$ , then all jobs of phase  $i + 1$  depend on  $b_i$ .

Since  $b_i$  is the job that finishes last at phase  $i$ , and all jobs of phase  $i + 1$  depend on it, then no jobs of phase  $i + 1$  are scheduled until all jobs of phase  $i$  are done. For  $1 \leq i \leq k$ , the jobs of phase  $i$  are restricted to  $2^{k-i+1} = m/2^{i-1}$  machines, thus the time to finish all jobs of phase  $i$  is at least  $(N+2-i)/2 = \Omega(N)$  (even with preemptions). Since there are  $\Omega(\log m)$  phases,  $C_{on} = \Omega(N \log m)$  (see Figure 3).

4								
4								
4								
3'								
3	3							
3	3							
2'	2	2	2	2				
2	2	2	2	2				
2	2	2	2	2				
1'	1	1	1	1				
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1

Fig. 3. A possible on-line assignment in the proof of Theorem 2

4	3	2	2	1	1	1	1	
4	3	2	2	1	1	1	1	
4	3	2	2	1	1	1	1	
3'	3	2	2	1	1	1	1	
2'	2	2	2	1	1	1	1	
1'	1	1	1	1	1	1	1	

Fig. 4. A possible optimal off-line assignment in the proof of Theorem 2

The optimal off-line algorithm schedules all  $b_i$  on the first machine, each  $b_i$  is scheduled at time  $i - 1$ . The jobs of phase  $i$  are scheduled as follows:  $m/2^{i-1}$  jobs are scheduled on machines  $1, \dots, m/2^{i-1}$  at time  $i - 1$ , all the other jobs are scheduled from time  $i$  till time  $N$  on machines  $m/2^i + 1, \dots, m/2^{i-1}$ . The jobs of phase  $k + 1$  are scheduled on machine 1 starting time  $\log_2 m$  (see Figure 4). We conclude that since  $C_{opt} = N$ , the competitive ratio is  $\Omega(\log m)$ .

To extend the proof for randomized algorithms we use the same sequence, but  $b_i$  is chosen uniformly at random among all jobs of phase  $i$ . Again, let  $P_i$  be the number of jobs that finish before  $b_i$  in phase  $i$ . The time after the jobs of phase  $i$  become available and before the next phase can start is at least  $(P_i + 1)/2^{k-i+1}$ . Since  $P_i$  gets the values  $0, \dots, (N + 2 - i)2^{k-i} - 1$  with equal probability,

$$E(P_i) = ((N + 2 - i)2^{k-i} - 1)/2.$$

Hence,

$$\begin{aligned} C_{on} &\geq \sum_{i=1}^k E(P_i + 1)2^{-k+i-1} + N - \log_2 m \\ &> \sum_{i=1}^k (N + 2 - i)/4 = O(N \log m). \end{aligned}$$

Since  $C_{opt} = N$  we conclude that the competitive ratio is  $\Omega(\log m)$ .

**Theorem 3.** *The competitive ratio of Greedy is  $O(\log m)$  for the restricted assignment model with consistent precedence constraints.*

*Proof.* For machine  $i$ , let  $A(i)$  be the set of jobs  $j$  that  $i \in M(j)$ . Denote the optimal off-line value by  $\lambda$ . We first prove the following Lemma:

**Lemma 1.** *The total idle time on a machine  $i$ , from the beginning till the last job in  $A(i)$  finishes its process (on any machine) is bounded by  $\lambda$ . (Some of this idle time may be after the last job on  $i$  is already completed).*

*Proof.* For each machine  $i$ , we build a chain of jobs in which each job is dependent on the previous job, and each time  $i$  is idle, one of the jobs in the chain is running. Since the total running time of jobs in the chain is at most  $\lambda$  (the optimal off-line algorithm can not run more than one job of the chain simultaneously), the total idle time of machine  $i$  would be also bounded by  $\lambda$ . We build the chain from the top, starting from the last job in the chain. If there is no idle time on machine  $i$ , the chain is empty and the lemma follows. Otherwise, we start the chain with the job in  $A(i)$  that finishes last, denote it by  $J_1$ . Assume that  $J_1, \dots, J_{q-1}$  are defined. If  $J_{q-1}$  has no predecessors, we finish the chain. Otherwise, let  $J_q$  be the predecessor of  $J_{q-1}$  that finishes last. Note that since all the chain consists of predecessors of  $J_1$  and the precedence constraints are consistent, all the jobs in the chain are also in  $A(i)$ . Assume that  $i$  is idle at time  $t$ , and no job in the

chain is running at time  $t$ . There is at least one job that finishes after time  $t$  ( $J_1$  for example). Since there is no job of the chain running at time  $t$ , all these jobs start running after time  $t$ . Let  $J_r$  be the first job of the chain that starts running after time  $t$ . All the predecessors of  $J_r$  finish before time  $t$  thus since  $i$  is idle at  $t$ ,  $J_r$  could be scheduled at time  $t$  or before. This is a contradiction to the definition of Greedy.

Note that it follows from Lemma 1 that the total idle time on a machine from the beginning till the last job that runs on this machine is completed is also bounded by  $\lambda$ .

**Lemma 2.** *Let  $l \geq 3\lambda$  be some time during the process of the algorithm. If the total running time of jobs (or parts of jobs) that run after time  $l$  is  $T_l$  then the total running time of jobs that run after time  $l - 3\lambda$  is at least  $2T_l$ .*

*Proof.* Let  $k_1 = \lceil \frac{T_l}{\lambda} \rceil$ . The optimal off-line uses at least  $k_1$  machines to run the jobs that the on-line runs after time  $l$ . Since the maximum running time is bounded by  $\lambda$ , these jobs start after time  $l - \lambda$ . For each machine  $i$  among the  $k_1$  machines, there is a job that is allowed to be scheduled on it and is scheduled after time  $l - \lambda$ , thus machine  $i$  has at most  $\lambda$  idle time from time  $l - 3\lambda$  till time  $l - \lambda$ . The total running time on  $i$  in this time period is at least  $\lambda$ . Summing for all machines the total running time is at least  $k_1\lambda$ , and adding the running times after time  $l$  we get a total of  $k_1\lambda + T_l \geq T_l + T_l = 2T_l$

Now, we can complete the proof of the theorem. Let  $T$  be the total running time of all jobs, note that  $T \leq m\lambda$ . Let  $k = \lfloor C_{on}/(3\lambda) \rfloor$ . We can assume without loss of generality that  $C_{on} \geq 3\lambda$ . Hence  $k \geq 1$ . Note that the competitive ratio  $r$  satisfies  $r = O(k)$ . Let  $T_j$  be the total running time of jobs after time  $C_{on} - 3j\lambda$ . According to Lemma 2,  $T_k$  satisfies  $T_k \geq 2^{k-1}T_1$  and according to Lemma 1,  $T_1$  satisfies  $T_1 \geq 2\lambda$ , this is correct since there is at least one machine that finishes at time  $C_{on}$ , and since the idle time on this machine is bounded by  $\lambda$ , this machine worked at least for a period of time  $2\lambda$  after time  $C_{on} - 3\lambda$ . Combining all observations together we get  $m\lambda \geq T \geq T_k \geq 2^{k-1}T_1 \geq 2 \cdot 2^{k-1}\lambda$ . Thus  $k = O(\log m)$ , and also  $r = O(\log m)$ .

## 4 Restricted Assignment with General Precedence Constraints

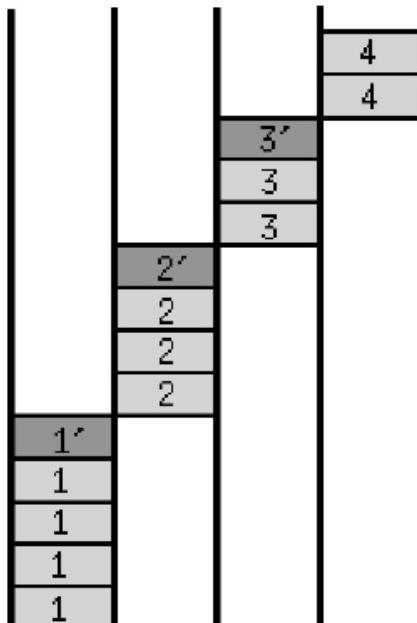
In this section we consider the restricted assignment model with general precedence constraints between jobs.

**Theorem 4.** *Any on-line scheduling algorithm for restricted assignment model with general precedence constraints has the competitive ratio of at least  $m$ . This is true even for preemptive algorithms versus non-preemptive adversary. Any randomized algorithm for the same problem has the competitive ratio of  $\Omega(m)$ . This is true even for randomized preemptive algorithms versus deterministic non-preemptive adversary.*

*Proof.* We first prove a lower bound on deterministic algorithms, and later extend it to randomized ones. We build the sequence according to the behavior of the on-line algorithm. Let  $N$  be an integer  $N \geq m$ , The optimal cost for the sequence would be  $N$ . The sequence contains  $m$  phases, in each phase, all jobs are restricted to a single machine. Phase 1 contains  $N$  unit jobs which are restricted to machine 1. Let  $b_1$  be the job from phase 1 that finishes last. We define the other phases recursively: In phase  $i$  ( $i \geq 2$ ), there are  $N - i + 1$  unit jobs which depend on the job  $b_{i-1}$ , and are restricted to machine  $i$ . We denote the job from phase  $i$  that finishes last by  $b_i$ .

The on-line does not schedule any job from phase  $i+1$  until all jobs of phase  $i$  are completed, because all jobs of phase  $i+1$  depend on  $b_i$ , thus the on-line has at most one working machine at a time (each job is restricted to a single machine) and the minimum possible on-line makespan is simply the sum of all running times:  $C_{on} \geq \sum_{i=1}^m (N - i + 1) = m(N - m/2 + 1/2)$  (see Figure 5).

The optimal off-line algorithm assigns each  $b_i$  at time  $i-1$ , and all other jobs of phase  $i$  are scheduled starting from time  $i$ , hence  $C_{opt} = N$  (see Figure 6). The competitive ratio is at least  $m - m^2/(2N) + m/(2N) > m - m^2/(2N)$ , for large values of  $N$ , this number approaches  $m$ .



**Fig. 5.** A possible on-line assignment in the proof of Theorem 4

1	2	3	4
1	2	3	4
1	2	3'	
1	2'		
1'			

**Fig. 6.** A possible optimal off-line assignment in the proof of Theorem 4

To extend the proof to randomized algorithms we use a similar sequence, which also has  $m$  phases, where phase  $i$  contains  $N-i+1$  jobs that are restricted to machine  $i$ , but here the job  $b_i$  for  $i = 1, \dots, m-1$  is chosen uniformly at random among all jobs of phase  $i$ . Let  $P_i$  be the position of  $b_i$ , which is the number of jobs from phase  $i$  that were completed before  $b_i$  was completed.  $P_i$  can get the values  $0, \dots, N-i$ , all with equal probabilities. For  $i \geq 2$ , the jobs of phase  $i$  are scheduled after at least  $P_{i-1} + 1$  jobs were completed at phase  $i-1$  and thus  $C_{on} = \sum_{i=1}^{m-1} (P_i + 1) + N - m + 1$ . Thus

$$E(C_{on}) \geq \sum_{i=1}^{m-1} (E(P_i) + 1) + N - m + 1 ,$$

since  $E(P_i) = (N-i)/2$  we get

$$\begin{aligned} E(C_{on}) &\geq (m-1)(N/2 + 1) - \sum_{i=1}^{m-1} i/2 + N - m + 1 \\ &= mN/2 - N/2 + m - 1 - m(m-1)/4 + N - m + 1 \\ &= mN/2 + N/2 - O(m^2) . \end{aligned}$$

Since  $C_{opt} = N$ , the competitive ratio is at least  $(m+1)/2 - O(m^2/N)$ , for large values of  $N$ , the lower bound approaches  $(m+1)/2 = \Theta(m)$ .

Both lower bounds are valid even with preemptions since we only consider finishing times of jobs, and not starting times.

**Theorem 5.** *The competitive ratio of Greedy is  $m$  for the restricted assignment model with precedence constraints.*

*Proof.* If all machines become idle, then there are no new jobs and the sequence is completed. Thus if  $C_{on} = T$ , there is at least one working machine during time  $T$ , the sum of all processing times is at least  $T$ , and  $C_{opt} \geq T/m$ , which gives the competitive ratio of  $m$ .

We can define greedy also for unrelated machines. The algorithm assign a job  $j$  to a machine  $i$  such that the running time of  $j$  on  $i$  is minimum over all  $i$ .

**Theorem 6.** *The competitive ratio of Greedy is  $m$  for the unrelated machines model with precedence constraints.*

*Proof.* Since the time that the optimal off-line uses to run each job is at least that of Greedy, we can imitate the proof of Theorem 5.

## References

1. Y. Azar, J. Naor, and R. Rom. The competitiveness of on-line assignments. *Journal of Algorithms*, 18(2):221–237, 1995. Also in *Proc. 3rd ACM-SIAM SODA*, 1992, pp. 203–210.
2. A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
3. E. Davis and J.M. Jaffe. Algorithms for scheduling tasks on unrelated processors. *J. Assoc. Comput. Mach.*, 28:712–736, 1981.
4. L. Epstein. Lower bounds for on-line scheduling with precedence constraints on identical machines. In *1st Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX98)*, volume 1444 of *LNCS*, pages 89–98, 1998.
5. R.L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.
6. R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17:416–429, 1969.
7. J.M. Jaffe. Efficient scheduling of tasks without full use of processor resources. *Theoretical Computer Science*, 12:1–17, 1980.
8. J. Sgall. On-line scheduling. In *A. Fiat and G. J. Woeginger, editors*, *Online Algorithms: The State of the Art*, volume 1442 of *LNCS*, pages 196–231. Springer-Verlag, 1998.
9. D. B. Shmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines on line. *SIAM J. on Computing*, 24:1313–1331, 1995.

# Scheduling Jobs Before Shut-Down

Vincenzo Liberatore<sup>1</sup>

UMIACS, A. V. Williams Building,  
University of Maryland,  
College Park, MD 20742,  
U.S.A.  
[vliberatore@acm.org](mailto:vliberatore@acm.org)

**Abstract.** Distributed systems execute background or alternative jobs while waiting for data or requests to arrive from another processor. In those cases, the following *shut-down scheduling problem* arises: given a set of jobs of known processing time, schedule them on  $m$  machines so as to maximize the total weight of jobs completed before an initially unknown deadline. We will present optimally competitive deterministic and randomized algorithms for shut-down scheduling. Our deterministic algorithm is parameterized by the number of machines  $m$ . Its competitive ratio increases as the number of machines decreases, but it is optimal for any given choice of  $m$ . Such family of deterministic algorithm can be translated into a family of randomized algorithms that use progressively less randomization and that are optimal for the given amount of randomization. Hence, we establish a precise trade-off between amount of randomization and competitive ratios. We also give a probabilistic analysis for the cases of uniform and exponential distributions. Finally, we report experimental results from trace-driven simulations.

## 1 Introduction

Internet traffic alternate lulls with spikes of extreme activity [4, 23]. As a result, Web performance is especially improved when operations are moved from peak periods to intervening lulls. For example, idle periods can be exploited by servers to speculatively disseminate data to dial-up clients, thus substantially reducing the latency experienced to retrieve subsequent documents [6]. Delays can stall the execution of a distributed query in a Web-based database system so as to trigger alternate queries or query plans [22].

*Shut-Down Scheduling.* In those architectures, if a job is preempted, it will not be resumed and any partially completed work is lost. Consequently, the following core optimization problem arises: a set of alternative or background jobs can be scheduled during a lull. A lull has unknown duration because it ends asynchronously when a message is received from a remote host. We will refer to such problem as *shut-down scheduling* because jobs execution is unpredictably interrupted. The off-line version of shut-down scheduling is a *maximum 0/1 multiple*

*knapsack problem* where all knapsacks have the same capacity. A book summarizes results in the theoretical and practical solution of the multiple knapsack problem [18]: it is strongly NP-hard [18], and a polynomial-time approximation scheme has been recently discovered [13]. Several authors have considered an on-line single knapsack problem where the deadline is known in advance, and jobs arrive on-line [15] [16]. The on-line knapsack problem can be regarded as the dual of shut-down scheduling and it is substantially an *admission control* problem [2] [7]. In general, shut-down scheduling is related to on-line *call control* [2] [7], *load balancing* [2] [10], and *bin packing* [2] [3] [8]. Scheduling with *machine breakdowns* has been considered as well [1] [11] [12]: jobs must be scheduled on  $m$  processors so as to complete in the presence of permanent or transient faults. Breakdowns differ from shut-down in several respects, as, for example, arrival times, objective function, job restart, redundant scheduling, and for the techniques and results of the analysis.

*Our Results.* We will present optimally competitive deterministic and randomized algorithms for shut-down scheduling. Randomized algorithms can be fully derandomized provided that there are sufficiently many machines. If there is only a small number  $m$  of machines, we will give an optimal deterministic algorithm CSM that is parameterized by  $m$ . The competitive ratio of CSM increases as  $m$  decreases, but, for any given choice of  $m$ , our CSM algorithm is optimal. We will also interpret CSM as a family of randomized algorithms that use progressively less randomization at the price of a worse competitive ratio. Our algorithm is optimal for any given choice of the amount of randomization and coincides with the optimal deterministic and randomized algorithms in the two extreme cases. Thus, such algorithm establishes a precise trade-off between randomization and competitive ratio. Randomized algorithms and lower bounds are transformed into deterministic algorithms and lower bounds by a technique that is simple and that might be more generally applicable to other scheduling problems. We report experimental results on Web trace simulations and indicate that a competitive algorithm indeed outperformed natural, but non-competitive strategies.

*Probabilistic Analysis.* We will also conduct a probabilistic analysis of algorithms for shut-down scheduling on  $m = 1$  machine. Probabilistic analyses of knapsack problems have been performed by several authors [5] [9] [14] [19] [21]. A probabilistic analysis was also performed for the on-line case [15] [16]. We will focus on shut-down scheduling and on the case when the deadline  $D$  is exponentially distributed. We will show a policy that maximizes the expected profit for the exponential distribution. We also present a shut-down schedule that breaks ties among jobs so as to minimize variance without worsening expected profit. Therefore, the resulting strategy is, in the parlance of portfolio theory, *E,V efficient* [17].

*Contents.* The paper is organized as follows. In §2, we introduce our notation for shut-down scheduling. In §3, we present competitive analyses and give optimal deterministic and randomized algorithm for shut-down scheduling. In §4, we

conduct a probabilistic analysis. In §5, we sketch some of the results of our simulations.

## 2 Preliminaries

In this section, we give definitions and notations for the shut-down scheduling problem. First, we introduce our notation for the  $m = 1$  machine problem. The *maximum 0/1 knapsack* problem is: given lengths  $l(i) \in \mathbb{N}$  ( $i \in [n] \stackrel{\text{def}}{=} \{1, 2, \dots, n\}$ ), profits  $p(i) \in \mathbb{N}$  ( $i \in [n]$ ), and a deadline  $D$ , find a subset  $J \subseteq [n]$  such that  $\sum_{i \in J} l(i) \leq D$  that maximizes  $\sum_{i \in J} p(i)$ . We will now introduce some notation. The profit  $p(J)$  and length  $l(J)$  of a set  $J \subseteq [n]$  are defined in the obvious way:  $p(J) = \sum_{i \in J} p(i)$  and  $l(J) = \sum_{i \in J} l(i)$ . Let  $p^*(D) = \max_{J: l(J) \leq D} p(J)$  be the optimum 0/1 knapsack objective value. Let  $\pi = (\pi_1, \pi_2, \dots, \pi_k)$  be a  $k$ -permutation of  $[n]$ , define  $J_i = \{\pi_1, \pi_2, \dots, \pi_i\}$  ( $1 \leq i \leq k$ ) and  $p(\pi, D) = p(J_s)$  where  $s$  is the largest integer such that  $l(J_s) \leq D$ . Note that when  $k < n$  and  $D > l(J_k)$ , the machine will remain idle after the completion of the  $k$  scheduled jobs. Although it does not seem intuitive, some algorithms will in fact exploit the fact that only some of the jobs are scheduled. Finally, we will omit the reference to  $D$  in  $p^*(D)$  and  $p(\pi, D)$  when the deadline  $D$  is clear from the context.

We will consider a two-person zero-sum game which is based on the maximum 0/1 knapsack problem and which we call the *knapsack game*. In the knapsack game, all the values  $l(i)$  and  $p(i)$  ( $i \in [n]$ ) are known at the beginning, but the deadline  $D$  is not. The player  $G$  selects a permutation  $\pi$  of  $[n]$  and the player  $H$  chooses  $D$ . If  $p^*(D) > 0$ , the quantity  $v(\pi, D) \stackrel{\text{def}}{=} p(\pi, D)/p^*(D)$  will be  $G$ 's *payoff* corresponding to the strategies  $\pi$  and  $D$ . If  $p^*(D) = 0$ , we define  $G$ 's payoff to be one. The objective of  $G$  is to maximize its payoff in the game. Since  $G$ 's payoff is always at most one, we can assume without loss of generality that  $D \geq \min_{i \in [n]} l(i)$ . Notice that  $G$  has at most  $n!$  strategies and  $H$  has at most  $2^n$  strategies, so that, for a given  $n$ , the knapsack game is a finite matrix game. We interpret the knapsack game as an on-line problem as follows. We have a set of  $n$  jobs numbered from 1 to  $n$ . Each job has a profit  $p(i)$  and it takes  $l(i)$  units of time to be completed. The on-line algorithm  $G$  starts to schedule jobs on one machine according to some ordering  $\pi$ . At time  $D$ , the adversary shuts the machine down, and  $G$  gains the values of all the jobs completed before  $D$ . A strict *competitive ratio* is an upper bound to the inverse of the game value. We do not allow additive terms in the competitive ratio because the game is finite. We remark the difference among the following quantities relative to an (on-line) algorithm:

**Profit** Total profit of jobs completed before the deadline

**Payoff** Ratio of the algorithm's profit over the adversary's. The payoff is relative to the chosen strategies for the on-line algorithm and for the adversary.

**Game Value** Best payoff an on-line player can achieve.

**Competitive Ratio** An upper bound on the inverse of the game value.

It can be noticed that the competitive ratio is defined in terms of inverse of game values, which is the correct choice in maximization on-line games [2]. We observe that the knapsack game is trivial if all  $p(i)$ 's are equal (choose the jobs in non-decreasing length order) or if all  $l(i)$ 's are equal (choose the jobs in non-increasing profit order). In the more general scenario, we will assume that a job can be scheduled on any one of  $m$  machines that run at the same speed. The adversary will shut all machines down at the deadline  $D$ .

Henceforth, we will use natural logarithms because they simplify notation and derivatives. Of course, the logarithm base does not alter the order of asymptotic bounds. Finally, we introduce some quantities that will be fundamental to the analysis below. Define  $V = \max_{i \in [n]} \{p(i)\} / \min_{i \in [n]} \{p(i)\}$  as the ratio of the largest to the smallest profit. Another important quantity is  $L$ , the number of distinct length values in the job set, that is,  $L = |\{l(j) : j \in [n]\}| \leq n$ . Finally, we define the *critical number of machines*  $\mu = \min\{L - 1, \ln V\}$ .

### 3 Competitive Analysis

In this section, we conduct competitiveness analysis for the shut-down scheduling problem.

#### 3.1 Randomized Algorithms

We present strongly competitive randomized algorithms for shut-down scheduling. Here, we will focus on the case when  $L, V \neq O(1)$ , and we will obtain different competitive ratios depending on the relative growth rate of  $L$  and  $V$ . We begin with a lower bound on the case of  $m = 1$  machine

**Lemma 1.** *No randomized algorithm for the knapsack game can be better than  $\Omega(L)$ -competitive when  $V = \Omega(2^L)$  and better than  $\Omega(\log V)$ -competitive when  $V = o(2^L)$ .*

The proof will exploit the minimax principle [2].

*Proof (Sketch).* Let  $\rho = 1/\sqrt[L-1]{V}$ ,  $l(i) = n + i - 1$  and  $p(i) = \lfloor p(1)\rho^{1-i} \rfloor$  for all  $i \in [n]$ . Notice that  $p(1) \leq p(2) \leq \dots \leq p(n) \leq V$ , so that the ratio of the largest to the smallest value is indeed bounded by  $V$ . The two fundamental points of the proof are the following. First, if  $D \leq 2n - 1$ , at most one job can be scheduled before the deadline. If the on-line player guesses the right job, its payoff is one. If it guesses a job that is longer than the deadline, its payoff is naught. Finally, if it guesses a job that is shorter than the deadline, its payoff is limited by the exponential growth of profits. The second point is to use the minimax principle as follows. Let  $c = n(1 - \rho) + \rho$ . We can show a probability distribution over  $D$  that forces any deterministic on-line strategy to have payoff  $O(1/c)$ . By the minimax principle, the value of the game is  $O(1/c)$ , and so the competitive ratio of a randomized on-line algorithm is  $\Omega(c) = \Omega(n(1 - \rho))$ . An asymptotic analysis of  $c$  completes the proof.  $\square$

We now show that the same lower bound holds for an arbitrary number  $m$  of machines.

**Corollary 2.** *No randomized algorithm for the shut-down scheduling on  $m$  machines can be better than  $\Omega(L)$ -competitive when  $V = \Omega(2^L)$  and better than  $\Omega(\log V)$ -competitive when  $V = o(2^L)$ .*

*Proof.* The proof is a reduction to the case of  $m = 1$  machine. Consider the same counterexample as in Lemma 1 on  $n'$  jobs and replicate each job for  $m$  times, so that the total number of jobs is now  $n = n'm$ . The number  $L$  of length classes remains unchanged. Again, at most one job can complete on any machine when  $D \leq 2n - 1$ . We will show how to convert any randomized algorithm for the  $m$  machine instance into a randomized algorithm for the original one-machine problem so that the two schedules achieve the same payoff. If  $D \leq 2n - 1$ , any randomized strategy for  $m$  machines is completely characterized by the expected number  $f_i$  of machines starting a job of length  $n + i - 1$ . Let  $h$  be the index with  $D = l(h)$ . By linearity of expectation, the on-line expected profit is  $\sum_{i=1}^h f_i p(i)$ . Meanwhile, the adversary's profit is  $mp(h)$ , so that the on-line expected payoff is  $\sum_{i=1}^h f_i p(i)/(mp(h))$ . Consider an on-line algorithm for the one machine instance that schedules job  $i$  with probability  $f_i/m$ . Its expected payoff is exactly the same as the  $m$  machine algorithm for any choice of deadline  $D \leq 2n - 1$ . Hence, the same lower bound as in Lemma 1 applies, and the proof is complete.  $\square$

If  $m > 1$  and all  $l(i)$ 's are equal, then shut-down scheduling is trivial (schedule jobs in non-increasing profit order). We turn to the case when the  $p(i)$ 's are equal, and show an  $O(1)$ -competitive algorithm. Such algorithm is an intermediate step to solve the case of general profits. First, notice that when all profits are equal, our objective is to maximize the number of completed jobs. Define the *canonical job scheduling* algorithm for a set  $C \subseteq [n]$  as a list scheduling algorithm 10 that orders the jobs from the shortest to the longest.

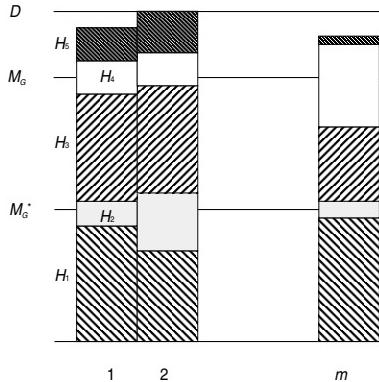
**Lemma 3.** *Let  $C \subseteq [n]$  be a set of jobs. The canonical schedule of  $C$  completes at least  $1/5$  of the jobs completed by any other algorithm before the deadline  $D$ .*

Define the *load* of a machine as the total length of jobs completed on that machine before the deadline and the *makespan* as the maximum load of any one machine. The proof will exploit a result for load balancing of permanent jobs 10.

*Proof.* The proof is organized as follows. We partition the jobs executed by the optimum into five classes, depending on their starting and completion time with respect to the deadline  $D$  and the makespan of the canonical schedule. Then, we show that no class contains more jobs than those completed by the canonical schedule before the deadline  $D$ . Hence, the canonical schedule completes at least  $1/5$  of the jobs completed by the optimum, which will complete the proof.

Assume without loss of generality that jobs are numbered in non-decreasing order of length, that is,  $l(j+1) \geq l(j)$  for  $j = 1, 2, \dots, n-1$ . Let  $G$  be the set

of jobs completed by the canonical schedule before  $D$  and let  $M_G$  the makespan of  $G$ , that is the time the last job in  $G$  is completed. By definition,  $M_G \leq D$ . Observe that initially  $G$  starts by assigning one job in  $[m]$  to each machine. Let  $H$  be a the largest set of jobs completed before  $D$ . Suppose first that  $|H| \leq m$ . Then,  $D$  is at least the length of the longest job in  $H$ , which is at least  $l(|H|)$ . Hence,  $G$  completes at least one job on at least  $|H|$  machines and the claim is proven. Assume now  $|H| > m$ , so that  $D \geq l(|H|) \geq l(m)$ . Hence,  $|G| \geq m$ . We now make the following definition: if  $X \subseteq [n]$  is a set of jobs, then  $M_X^*$  is the minimum makespan to complete  $X$ . Observe that if  $X \subseteq Y$ , then  $M_X^* \leq M_Y^*$ . Analogously, if  $X, Y \subseteq [n]$  and if there is a one-to-one mapping  $f : Y \rightarrow X$  with  $l(j) \geq l(f(j))$  for all  $j \in Y$ , then  $M_X^* \leq M_Y^*$ . Let  $M_G^*$  be the earliest time when  $G$  can be completed. A load balancing results claims that  $2M_G^* > M_G$ . Schedule  $H$  on  $m$  machine so that the schedule completes before time  $D$  and partition  $H$  into five subsets according to such schedule as follows (Figure 1 gives an example of such partition).



**Fig. 1.** A partition of the optimal set  $H$  of jobs according to the optimal makespan  $M_G^*$  and the actual makespan  $M_G$  of the on-line algorithm.

- The subset  $H_1 \subseteq H$  is the set of jobs that complete before time  $M_G^*$ . We claim that  $|H_1| \leq |G|$ . Suppose by contradiction  $|H_1| > |G|$ . Take any proper subset  $H' \subset H_1$  with  $|H'| = |G|$  elements. Since  $G$  consists of the  $|G|$  shortest jobs, there is a one-to-one mapping  $f : H' \rightarrow G$  with  $l(j) \geq l(f(j))$  for all  $j \in H'$ . Hence, the optimal makespan  $M_{H'}^*$  of  $H'$  is not smaller than the optimal makespan  $M_G^*$  of  $G$ . Therefore,  $M_G^* \leq M_{H'}^* \leq M_H^*$  and a contradiction is reached. Therefore, we conclude  $|H_1| \leq |G|$ .
- The subset  $H_2 \subseteq H$  is the set of jobs that start before time  $M_G^*$  and complete after time  $M_G^*$ . The set  $H_2$  contains at most one job per machine, so that  $|H_2| \leq m \leq |G|$ .

- The subset  $H_3 \subseteq H$  is the set of jobs that starts after time  $M_G^*$  and complete after time  $M_G$ . Since  $M_G - M_G^* < M_G^*$ , we conclude that  $|H_3| \leq |G|$  with an argument similar to  $H_1$ .
- The subset  $H_4 \subseteq H$  is the set of jobs that starts before time  $M_G$  and complete after time  $M_G$ . The set  $H_4$  contains at most one job per machine and so  $|H_4| \leq m \leq |G|$ .
- The subset  $H_5 \subseteq H$  is the set of jobs that start after  $M_G$ . Notice that  $D - M_G < l(|G| + 1)$  or else  $G$  would have scheduled one more job. Hence,  $H_5$  can contain at most  $|G|$  documents because job  $|G| + 1$  does not fit in the allotted time  $D - M_G < l(|G| + 1)$ .

Notice that  $H_1, H_2, \dots, H_5$  give indeed a partition of  $H$ . We conclude that  $|H| \leq 5|G|$ , which proves the lemma.  $\square$

**Corollary 4.** *The canonical schedule is 5-competitive for shut-down scheduling on any number  $m$  of machines when  $p(i) = 1$  for all jobs  $i$ .*

The canonical schedule algorithm easily generalizes to the case of arbitrary profits by using the CRS techniques. Partition the job set into  $O(\log V)$  profit classes such that no job is more than  $O(1)$  times as profitable as any other job in the same class. Then, extract a profit class at random and execute jobs only from that class. However, if  $V = \Omega(2^L)$ , then jobs are partitioned according to their length in such a way that a job class contains only jobs of the same length. We conclude that

**Theorem 5.** *The best randomized algorithm for shut-down scheduling is  $\Theta(L)$ -competitive when  $V = \Omega(2^L)$  and  $\Theta(\log V)$ -competitive when  $V = o(2^L)$ .*

A consequence of the matching upper and lower bounds is that if we change the number  $m$  of machines, we do not help nor hamper the competitive ratio of randomized algorithms.

### 3.2 Deterministic Algorithms

We now turn to deterministic algorithms. It is helpful during the discussion to refer to table 1 which summarizes our results. First, we argue that if there is

	$V = o(2^L)$	$V = \Omega(2^L)$
$m \leq \mu$	$\Theta(m\rho)$	$\Theta(m\rho)$
$m > \mu$	$\Theta(\log V)$	$\Theta(L)$

**Table 1.** Competitive ratios of the best deterministic algorithm for the  $m$  machine knapsack game, where  $L$  is the number of length classes,  $V$  is the ratio of largest and smallest profit,  $\mu = \min\{L - 1, \ln V\}$  is the critical number of machines, and  $\rho = \sqrt[m]{V}$ .

a sufficiently large number  $m > \mu$  of machines, then, we can find deterministic

algorithms that match the randomized lower bound. We derandomize the CRSlog algorithm as follows. If we have  $m \geq \mu + 1 \geq \ln V + 1$  machines, we can assign  $m' = \lfloor m/(\ln V + 1) \rfloor$  machines to process jobs in each profit class according to the canonical schedule. Roughly speaking, the derandomized version translates the probability of executing jobs in class  $C_i$  into the fraction of machines assigned to class  $C_i$ . It is critical that profit classes be disjoint sets, as otherwise a job would have to be scheduled on more than one machine.

**Lemma 6.** *The derandomized version of the CRSlog algorithm is  $O(\log V)$ -competitive for shut-down scheduling on  $m \geq \ln V + 1$  machines.*

*Proof.* At any time, the algorithm has completed at least  $1/5$  of the jobs in a certain class that are completed by any other algorithm that uses  $m'$  machines for that class. Hence, the adversary completes in each class at most  $5m/m' = O(\log V)$  jobs more than the derandomized CRSlog algorithm, and, on each job, it earns less than  $e$  times as much as the derandomized CRSlog. Thus, such algorithm is  $O(\log V)$ -competitive.  $\square$

We can analogously derandomize the  $O(L)$ -competitive algorithm as long as we have  $m \geq \mu + 1 \geq L$  machines. It remains to establish deterministic competitive ratios for  $m \leq \mu$  machines. In this case, Corollary 2 is tight for randomized algorithms, but gives a weak lower bound for deterministic algorithms. Intuitively, the weakness of Corollary 2 stems from the fact that it is not always possible to execute simultaneously all deterministic strategies that compose a randomized algorithm if only few machines are available. Define  $\rho \stackrel{\text{def}}{=} \sqrt[m]{V}$  (such notation is independent of that in Lemma 1) and notice that  $\rho > 1$ . We will frequently use the equality  $\ln \rho = (\ln V)/m$  and  $m = \ln V / \ln \rho$ .

**Lemma 7.** *If  $m \leq \min\{L - 1, \ln V\}$ , then no deterministic algorithm can be better than  $\Omega(mp)$ -competitive.*

Notice that  $mp = \omega(m \log \rho) = \omega(\log V)$ . On the other hand, if  $V = \Omega(2^L)$ , then  $\rho = \Omega(2^{L/m}) = \omega(L/m)$ , and so  $mp = \omega(L)$ . Hence, Lemma 7 dominates Corollary 2 when  $m \leq \mu$ .

*Proof (Sketch).* The proof is based on an instance with the property that the adversary will be able to choose a bad deadline for any on-line algorithm. The instance consists of  $m + 1$  classes of  $m$  identical jobs such that jobs in class  $i$  are  $\rho$  times more valuable than jobs in class  $i - 1$ . Job lengths are chosen in such a way that only one job can complete on any one machine, which is similar to the length distribution of Lemma 1. Since there are more classes than machines, the on-line algorithm does not schedule any job from a certain class. The adversary chooses the deadline so that the optimum strategy schedules jobs only from that class, while the on-line algorithm achieves a small profit. We will now give some details of the arguments.

*Set-up.* The proof is based on an instance where there are  $m$  identical jobs of profit  $\lfloor p_0 \rho^i \rfloor$  ( $i = 0, 1, 2, \dots, m$ ) for some minimal profit  $p_0$ . Hence, the total number of jobs is  $n = m(m+1)$ . Notice that  $L = m+1 > m$ . Observe that the minimum profit is  $p_0$  and the maximum profit is no more than  $p_0 V$ . A profit class is a set of jobs with the same profit. We will think of profit classes as ordered by the profit of the jobs they contain. A job of profit  $p_0 \rho^i$  has length  $2m+i$ . If  $D \leq 3m$ , then at most one job can complete on any one machine.

*Holes.* Since the number of profit classes is  $m+1$ , there is at least one profit class from which no job is completed before the deadline  $D \leq 3m$ . We will say that a *hole* is a maximal non-empty sequence of profit classes with the property that no job has been scheduled from any class in the hole. Clearly, there is at least one hole. If the first class  $C_0$  is in a hole, then the adversary will set  $D = 2m$ , and the on-line algorithm achieves no profit. Therefore, we can assume from now that the first class is not in a hole without loss of generality. Suppose that the holes are  $\mathcal{H}_1 \prec \mathcal{H}_2 \prec \dots \prec \mathcal{H}_l$ . Let  $k_i$  be the number of jobs scheduled from the class immediately preceding hole  $\mathcal{H}_i$ . We claim that there is at least one hole  $\mathcal{H}_i$  for which  $k_i \leq |\mathcal{H}_i| + 1$ . Suppose that this is not true. Then, denote by  $\nu$  the number of classes that are not immediately followed by a hole and observe that the total number of jobs is at least  $\nu + 2l + \sum_{i=1}^l |\mathcal{H}_i| = m + l > m$ , which is a contradiction. The adversary chooses a deadline equal to the maximum length of a job in a hole  $\mathcal{H}$  such that  $|\mathcal{H}|+1$  is a bound on the number of jobs scheduled in the class immediately before the hole.

An analysis of payoffs concludes the proof.  $\square$

Such lower bound is matched by the following CSM (Canonical Schedule for  $m$  machines) algorithm. First, normalize job profits so that the minimum profit is one and the maximum profit is  $V$ . CSM divides the job set into  $m$  classes according to job profit, where class  $i$  consists of jobs of profit  $\rho^{i-1} \leq p(j) < \rho^i$ . CSM schedules jobs of class  $i$  on machine  $i$  according to the canonical schedule.

**Lemma 8.** *The CSM algorithm is  $O(m\rho)$ -competitive for the knapsack game on  $m$  machines.*

*Proof.* Define the  $h$ th profit class as  $C_h = \{j : \rho^{h-1} \leq p(j) < \rho^h\}$ . Let  $x_h$  be the number of jobs in class  $h$  scheduled by the optimum before the deadline  $D$ . Hence, the optimum profit from class  $h$  is less than  $x_h \rho^h$ . The CSM algorithm schedules at least a  $x_h/(5m)$  jobs in class  $h$ , so that its profit is at least  $x_h \rho^{h-1}/(5m)$ . Hence, CSM has a payoff of at least

$$\frac{\sum_{h=1}^m x_h \rho^{h-1}}{m \sum_{h=1}^m x_h \rho^h} \geq \frac{\sum_{h=1, x_h \neq 0}^m x_h \rho^{h-1}}{m \sum_{h=1, x_h \neq 0}^m x_h \rho^h} \geq \frac{1}{5m} \frac{1}{\rho} \geq \frac{1}{5m\rho}.$$

The competitive ratio is the inverse of the payoff, and thus the proposition is proven.  $\square$

The lower bound and the CSM algorithm are summarized by:

**Theorem 9.** *The best deterministic algorithm for the knapsack game on  $m \leq \min\{L - 1, \ln V + 1\}$  machines is  $\Theta(m \sqrt[m]{V})$ -competitive.*

### 3.3 Reduced Randomization

The CSM algorithm can be translated into a randomized algorithm, where a random class of jobs is scheduled according to the canonical schedule. We will name the resulting randomized algorithm CSMr.

**Lemma 10.** *The CSMr algorithm is  $O(m\rho)$ -competitive for the shut-down scheduling problem.*

*Proof.* Since there are  $m$  profit classes, jobs in the same classes have profits that are within a factor of  $\rho$ , and the canonical schedules has a performance guarantee of 5, we obtain that the payoff of CSMr is at least  $1/(5m\rho)$ .  $\square$

Hence, CSMr gives a precise trade-off between randomization and competitive ratio. Indeed, if no randomization is allowed, then the best algorithm is  $\Theta(V)$ -competitive. As the amount of randomization increases to  $m$  strategies ( $m \leq \ln V$ ), performance improves as  $\Theta(m^{\sqrt[m]{V}})$ . Finally, when  $m = \ln V + 1$ , the best algorithm is  $\Theta(\log V)$ -competitive and, if  $V = o(2^L)$ , no further improvement stems from adding more machines. Meanwhile, we claim that CSMr achieves optimal performance.

**Theorem 11.** *The best randomized algorithm that is a distribution over only  $m$  deterministic strategies is  $\Theta(m\rho)$ -competitive.*

*Proof (Sketch).* Consider the proof of Lemma 7 and replace the number of machine starting a certain job class with the expected number of machines.  $\square$

## 4 Probabilistic Analysis

In this section, we will conduct probabilistic analyses of shut-down scheduling on  $m = 1$  machine. Let  $\pi = (\pi_1, \pi_2, \dots, \pi_n)$  be a permutation of  $[n]$  and  $J_i = \{\pi_1, \pi_2, \dots, \pi_i\}$ . Then,

$$E[p(\pi)] = \sum_{i=1}^n p(J_i) Pr[l(J_i) \leq D < l(J_{i+1})] = \sum_{i=1}^n p(\pi_i) Pr[D \geq l(J_i)]. \quad (1)$$

Our objective is to find a permutation  $\pi$  that maximizes (1). A corresponding decision problem is to find a permutation  $\pi$  such that  $E[p(\pi)] \leq \bar{p}$  for some given  $\bar{p}$ . Such decision problem is easily seen to be NP-complete as it reduces to a knapsack problem when there is a  $t$  with  $Pr[D = t] = 1$ . First, we give a general optimality criterion.

**Lemma 12.** *If a permutation  $\pi$  maximizes (1), then, for all  $i \in [n - 1]$ ,*

$$p(\pi_i) Pr[l(J_i) \leq D < l(J_{i+1})] \geq p(\pi_{i+1}) Pr[l(J_{i-1}) + l(\pi_{i+1}) \leq D < l(J_{i+1})].$$

*Moreover, if  $\pi$  maximizes (1) and*

$$p(\pi_i) Pr[l(J_i) \leq D < l(J_{i+1})] = p(\pi_{i+1}) Pr[l(J_{i-1}) + l(\pi_{i+1}) \leq D < l(J_{i+1})],$$

*then the permutation  $\pi'$  obtained by exchanging  $\pi_i$  and  $\pi_{i+1}$  is also optimal.*

*Proof (Sketch).* If this were not so, exchange jobs  $\pi_i$  and  $\pi_{i+1}$  to increase the profit. Analogously, if equality holds, the profit remains unchanged, and thus optimal.  $\square$

A simple corollary is that if all  $p(i)$ 's are equal, then the optimal solution is to arrange jobs in increasing order of length, independently of the distribution of  $D$ . Another simple consequence is that if  $D$  is uniformly distributed in an interval  $[0, A]$  of the real line with  $A \geq l([n])$ , then the optimal permutation is to arrange jobs in non-increasing *profit density*  $p(i)/l(i)$  (*PD-order*). The result for the uniform distribution also follows by noticing that, under such distribution, the objective (II) defines a weighted completion time problem, and we can apply Smith's rule [20].

We now turn to the case when the deadline  $D$  is extracted according to an exponential distribution with rate  $\lambda$ . The exponential distribution models the case when client requests arrive according to a Poisson process, and each request terminates a lull. First, recall that, if  $D$  is exponentially distributed, we have  $Pr[D \geq t] = e^{-\lambda t}$ . Then, expression (II) becomes

$$E[p(\pi)] = \sum_{i=1}^n p(\pi_i) e^{-\lambda l(J_i)}. \quad (2)$$

Define the *exponential density* of job  $i$  as the ratio  $d_e(i) = p(i)/(e^{\lambda l(i)} - 1)$ . The *Exponential Profit Density* (EPD) algorithm arranges jobs in non-increasing order of exponential profit density. We will say that a permutation is in *EPD-order* if its jobs are in non-increasing order of exponential profit density.

**Theorem 13.** *If  $Pr[D \geq t] = e^{-\lambda t}$ , then a permutation maximizes (2) if and only if it is in EPD-order.*

*Proof.* If a permutation is optimal, then the optimality condition of Lemma 12 implies that it is in EPD-order. Conversely, assume that the identity is an optimal EPD permutation, and suppose we exchange two terms  $h$  and  $h+1$  with the same exponential value density. Lemma 12 implies that the new permutation is optimal as well. Any permutation in EPD-order can be obtained by a finite exchange of jobs with the same exponential profit density, and the proposition is proven.  $\square$

The previous theorem suggests that in some sense lengths are exponentially more important than values for an exponential distribution. On the other hand, an exponential distribution can be approximated by a uniform distribution when  $\lambda$  is large ( $\lambda \geq l([n])$ ), in which case we can show that PD is within 1.1312 of the optimum.

We observe that there are in general several scheduling strategies in PD-order (EPD-order). Although any such strategy maximizes the expected profit, we will show that some optimal strategies have smaller variance than others. Variance analysis is based on the optimality conditions and on the following

**Lemma 14.** *If  $\pi$  is an optimal permutation that has minimum  $\text{Var}[p(\pi)]$  among all optimal permutations and  $p(\pi_i)\Pr[l(J_i) \leq D < l(J_{i+1})] = p(\pi_{i+1})\Pr[l(J_{i-1}) + l(\pi_{i+1}) \leq D < l(J_{i+1})] \neq 0$ , then  $p(\pi_i) \leq p(\pi_{i+1})$ .*

*Proof (Sketch).* Since  $\pi$  is optimal, Lemma 12 holds. Hence, we can only exchange jobs for which the equality condition holds. Furthermore,  $p(\pi)$  is a constant among all optimal permutation, so that minimizing the variance is tantamount to maximizing the second moment  $E[p^2(\pi)]$ . Therefore, we seek optimality conditions for the problem where  $p(j)$  is replaced by  $-p^2(j)$ , subject to the constraints given by Lemma 12. Such optimality conditions are found by an exchange argument and the lemma is proven.  $\square$

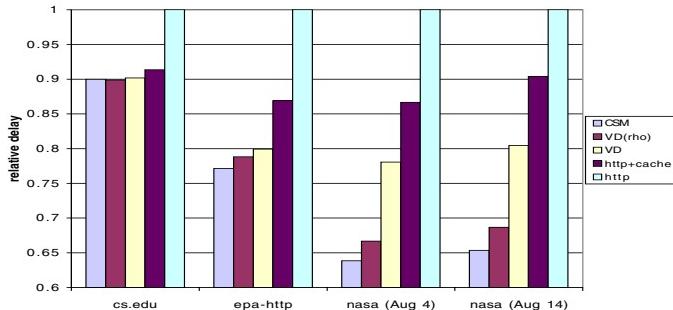
It can be seen that a tie breaking procedure for the uniform and exponential distribution is to favor shorter jobs. Indeed, suppose that job  $i$  and  $i + 1$  have the same (exponential) profit density and  $p(i) \leq p(i + 1)$ . Then,  $l(i) \leq l(i + 1)$ . Hence, the optimal strategy that minimizes risk is to arrange jobs in PD-order (EPD-order) and break ties by scheduling shortest jobs first.

## 5 Simulations

In this section, we sketch the set-up and results of our simulations. We postpone a complete account to the full paper. We simulated speculative data dissemination during server idle periods. Simulation is based on four Web server traces. The base server bandwidth value depends on the server load and is 8KB/s for cs.edu, 16KB/s for epa-http and 64KB/s for NASA. We assume that each client has an *extension cache* [6] of moderate size to keep both requested and disseminated documents. The major performance measure is the average *delay*. The delay is the time for a client to receive the requested document, and it includes a fixed network latency, the transmission time (document size over client bandwidth), and the time spent in the server queue. We simulated five strategies:

1. Traditional http with no client cache, which is our baseline strategy.
2. The http protocol with extension [6] client caches of 128 KB.
3. Profit density (PD).
4. PD(rho), which is similar to PD but takes into account the packet drop probability in estimating document profits, and
5. CSM for  $m = 1$  machine, which sends documents from the shortest to the longest.

Figure 2 compares delays normalized to the baseline http value. The CSM algorithm outperformed PD in all traces, except cs.edu, where the two algorithms have nearly equal performance. The PD algorithm performed better if it uses packet drop probabilities (PD(rho)), but even so, it was almost always outperformed by CSM. Both CSM and PD consistently improved over the case when no speculative dissemination is executed.



**Fig. 2.** Relative delays of traditional http with and without caches, PD with no knowledge of the packet drop probability (PD), PD with perfect knowledge of the packet drop probability (PD(rho)), and CSM ( $m = 1$ ). Sum of individual document latencies is normalized to traditional http with no client cache. Results for four Web server traces are reported.

## Acknowledgments

We are grateful to Brian Davison, Kevin Christian, Bala Kalyanasundaram, Samir Khuller, and Xiao-Tong Zhuang for helpful discussions, and to an anonymous referee for his comments and corrections. The author was partially supported by grant CCR-9501355.

## References

- [1] Susanne Albers and Guenter Schmidt. Scheduling with unexpected machine breakdowns. Technical Report MPI-I-98-1-021, Max-Planck-Institut Fuer Informatik, 1998.
- [2] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
- [3] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*, chapter 2. PWS, Boston, 1997.
- [4] Mark Crovella and Paul Barford. The network effects of prefetching. In *Proceedings of IEEE INFOCOM*, 1998.
- [5] Gianfranco D'Atri and Claude Puech. Probabilistic analysis of the subset-sum problem. *Discrete Appl. Math.*, 4(4):329–334, 1982.
- [6] Li Fan, Pei Cao, and Quinn Jacobson. Web prefetching between low-bandwidth clients and proxies: Potential and performance. In *Proceedings ACM SIGMETRICS '99*, pages 178–187, 1999.
- [7] Juan A. Garay, Inder S. Gopal, Shay Kutten, Yishay Mansour, and Moti Yung. Efficient on-line call control algorithms. *J. Algorithms*, 23(1):180–194, 1997.
- [8] M. R. Garey, R. L. Graham, and J. D. Ullman. An analysis of some packing algorithms. In *Combinatorial algorithms (Courant Comput. Sci. Sympos., No. 9, 1972)*, pages 39–47. Algorithmics Press, New York, 1973.

- [9] Andrew V. Goldberg and Alberto Marchetti-Spaccamela. On finding the exact solution to a zero-one knapsack problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 359–368, 1984.
- [10] R. L. Graham. Bound for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.
- [11] Bala Kalyanasundaram and Kirk Pruhs. Fault-tolerant scheduling. In *Proceedings of the Twentysixth Annual ACM Symposium on the Theory of Computing*, pages 115–124, 1994.
- [12] Bala Kalyanasundaram and Kirk Pruhs. Fault-tolerant real-time scheduling. In *Algorithms—ESA '97 (Graz)*, pages 296–307. Springer, Berlin, 1997.
- [13] Hans Kellerer. A polynomial time approximation scheme for the multiple knapsack problem. In *Proceedings of The 2nd International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, 1999.
- [14] George S. Lueker. On the average difference between the solutions to linear and integer knapsack problems. In *Applied probability—computer science: the interface, Vol. I (Boca Raton, Fla., 1981)*, pages 489–504. Birkhäuser Boston, Boston, Mass., 1982.
- [15] George S. Lueker. Average-case analysis of off-line and on-line knapsack problems. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 179–188, 1995.
- [16] A. Marchetti-Spaccamela and C. Vercellis. Stochastic on-line knapsack problems. *Math. Programming*, 68(1, Ser. A):73–104, 1995.
- [17] Harry M. Markowitz. *Portfolio Selection. Efficient Diversification of Investments*. Blackwell, New Haven, 1970.
- [18] Silvano Martello and Paolo Toth. *Knapsack problems*. John Wiley & Sons Ltd., Chichester, 1990.
- [19] M. Meanti, A. H. G. Rinnooy Kan, L. Stougie, and C. Vercellis. A probabilistic analysis of the multiknapsack value function. *Math. Programming*, 46(2 (Ser. A)):237–247, 1990.
- [20] Wayne E. Smith. Various optimizers for single-stage production. *Naval Res. Logist. Quart.*, 3:59–66, 1956.
- [21] Krzysztof Szkatuła and Marek Libura. On probabilistic properties of greedy-like algorithms for the binary knapsack problem. In *Stochastics in combinatorial optimization (Udine, 1986)*, pages 233–254. World Sci. Publishing, Singapore, 1987.
- [22] Tolga Urhan, Michael Franklin, and Laurent Amsaleg. Cost-based query scrambling for initial delays. In *ACM SIGMOD Intl. Conference on Management of Data (SIGMOD)*, pages 130–141, 1998.
- [23] Walter Willinger and Vern Paxson. Where Mathematics meets the Internet. *Notices of the AMS*, 45(8):961–970, September 1998.

# Resource Augmentation in Load Balancing

Yossi Azar<sup>1,\*</sup>, Leah Epstein<sup>1,\*\*</sup>, and Rob van Stee<sup>2,\*\*\*</sup>

<sup>1</sup> Dept. of Computer Science, Tel-Aviv University. [{azar,lea}@math.tau.ac.il](mailto:{azar,lea}@math.tau.ac.il)

<sup>2</sup> Centre for Mathematics and Computer Science (CWI). [Rob.van.Stee@cwi.nl](mailto:Rob.van.Stee@cwi.nl)

**Abstract.** We consider load balancing in the following setting. The on-line algorithm is allowed to use  $n$  machines, whereas the optimal off-line algorithm is limited to  $m$  machines, for some fixed  $m < n$ . We show that while the greedy algorithm has a competitive ratio which decays linearly in the inverse of  $n/m$ , the best on-line algorithm has a ratio which decays exponentially in  $n/m$ . Specifically, we give an algorithm with competitive ratio of  $1 + 1/2^{\frac{n}{m}(1-o(1))}$ , and a lower bound of  $1 + 1/e^{\frac{n}{m}(1+o(1))}$  on the competitive ratio of any randomized algorithm.

We also consider the preemptive case. We show an on-line algorithm with a competitive ratio of  $1 + 1/e^{\frac{n}{m}(1+o(1))}$ . We show that the algorithm is optimal by proving a matching lower bound.

We also consider the non-preemptive model with temporary tasks. We prove that for  $n = m + 1$ , the greedy algorithm is optimal. (It is not optimal for permanent tasks).

## 1 Introduction

Competitive analysis has been criticized for being too pessimistic. This worst case analysis sometimes fails to differentiate between algorithms whose performance is observed empirically to be very different. A general method to circumvent these shortcomings was introduced by KALYANASUNDARAM and PRUHS [13]: resource augmentation. For certain scheduling problems with unbounded competitive ratio, they show that it is possible to attain a good competitive ratio if the machines of the on-line algorithm are slightly faster than the machines of the off-line algorithm.

Resource augmentation has been applied to a number of problems. It was already used in the paper where the competitive ratio was introduced [20]: here the performance of some paging algorithms was studied, where the on-line algorithm has more memory than the optimal off-line one.

In several machine scheduling and load balancing problems [4, 8, 13, 14, 16, 18], the effect of adding more or faster machines has been studied.

---

\* Research supported in part by the Israel Science Foundation and by the United States-Israel Binational Science Foundation (BSF).

\*\* Part of the research was done while this author was visiting the Centre for Mathematics and Computer Science (CWI), supported by a grant from the Netherlands Organization of Scientific Research.

\*\*\* Research supported by the Netherlands Organization for Scientific Research (NWO), project number SION 612-30-002.

We consider the following load balancing problem. Jobs arrive on-line, where job  $j$  has a certain weight  $w_j$ . The job has to be assigned immediately to a machine, adding  $w_j$  to the machine's load. The on-line algorithm has  $n$  identical machines, and it is compared to an optimal offline algorithm which has  $m < n$  identical machines.

For a job sequence  $\sigma$  we write  $A_n(\sigma)$  for the maximum load of  $A$  on  $n$  machines when it is given this job sequence. Analogously, we write  $OPT_m(\sigma)$ . We denote the competitive ratio of an online algorithm  $A$  with  $n$  machines relative to an optimal offline algorithm with  $m$  machines by  $c_{m,n}(A)$ . Specifically,

$$c_{m,n}(A) = \max_{\sigma} \frac{A_n(\sigma)}{OPT_m(\sigma)}.$$

The classical case of  $n = m$  was considered in a series of papers [11, 12, 3, 15, 1]. The best upper bound is 1.923 due to ALBERS [1] and the best lower bound is 1.853 [10] based on [1]. The case  $n > m$  was introduced by BREHOB et al [5]. They showed that no matter how many machines the on-line algorithm has, it can never perform optimally:  $c_{m,n}(A) > 1$  for all  $n > m \geq 2$ . However, one may expect that for reasonable algorithms  $c_{m,n}(A)$  would approach 1 when  $t = n/m$  increases. In fact, [5] showed that the greedy algorithm has a competitive ratio which approaches 1 in a rate depending linearly on  $1/t$ .

In contrast, while the greedy algorithm has a competitive ratio which approaches 1 in a rate depending linearly on  $1/t$ , we design a non-greedy algorithm whose competitive ratio approaches 1 in a rate depending exponentially on  $t$ . More specifically, we give an algorithm of competitive ratio  $1 + \frac{1}{2^{t(1-o(1))}}$ . Moreover, we show that the competitive ratio of any on-line algorithm cannot decrease faster than exponentially in  $t$  by proving a lower bound of  $1 + \frac{1}{e^{t(1+o(1))}}$  on the competitive ratio of any on-line algorithm. We also show for  $n = 2m$  a lower bound of  $5/4$ .

We also consider the preemptive case. Here we view load as time. Each job may be assigned to one or more machines and time slots, where the time slots have to be disjoint. The assignment has to be determined completely at the arrival of a job. Using similar techniques as in [6, 7, 19] we prove a lower bound of  $1/(1 - (\frac{m-1}{m})^n) = 1 + \frac{1}{e^{t(1+o(1))}}$  on the competitive ratio of any randomized preemptive algorithm. We also show a matching upper bound by adapting the optimal preemptive algorithm of [7] to our problem.

We can also view time as a separate axis and not as the load axis. Here jobs arrive and depart at arbitrary times and the cost of an algorithm is the maximum load over time and machines. This model is called the temporary tasks model (the case where jobs only arrive is called the permanent tasks model). It was proved in [2] that for  $n = m$  the greedy algorithm, which is  $2 - 1/m$  competitive, is optimal for this model. We show that if  $n$  is just slightly larger than  $m$ , i.e.,  $n = m + 1$ , then greedy which is  $2 - 2/(m + 1)$  competitive is also optimal. Note that the results in [1] implies that the greedy algorithm is not optimal in general for permanent tasks also for  $n > m$ .

## 2 Permanent Tasks

In this section we check the growth of the competitive ratio as a function of  $t = n/m$ . We start with the competitive ratio of the greedy algorithm. This algorithm was first given by GRAHAM [11], and assigns each new job to the least loaded machine. The following lemma is shown in [5] using a similar analysis as in [11]:

**Lemma 1.** *The competitive ratio of the greedy algorithm is  $1 + \frac{m-1}{n}$ .*

The above theorem implies a competitive ratio which is a linear function in  $1/t$ . Surprisingly, we can give an algorithm called *Buckets* which has a competitive ratio  $1 + 1/2^{t(1-o(1))}$ .

### 2.1 Algorithm Buckets

For describing the algorithm *Buckets* we assume that  $t > 3$ . (If  $t \leq 3$  we use the greedy algorithm.) Let  $0 < \varepsilon < 1$  some parameter to be fixed later. We partition all machines into buckets:  $k = \lfloor t - \frac{2}{\varepsilon} \rfloor$  small buckets, each of which contains  $m$  machines, and one big bucket that contains all other machines. Note that the big bucket contains at least  $\frac{2m}{\varepsilon}$  machines.

Algorithm *Buckets* maintains a value  $\lambda$ . Denote by  $\lambda_i$  the value of  $\lambda$  after the arrival of  $i$  jobs and by  $OPT_i$  the optimal load after  $i$  jobs. The algorithm consists of phases. During a phase  $j$ , the algorithm can use only the big bucket and the small bucket number  $j \bmod k$ . We assign the first job to the first small bucket and initialize  $\lambda_1 = w_1$ . We modify  $\lambda$  only when a new phase starts while keeping the following two invariants on  $\lambda$ :

- $\max_{j \leq i} w_j \leq \lambda_i$
- $(2 - \varepsilon)OPT_i \geq \lambda_i$

On arrival of a job  $i$  (starting from  $i = 2$ ), we do the following: If  $w_i \leq \lambda_{i-1}/2$  assign  $i$  greedily to the least loaded machine in the big bucket. If  $\lambda_{i-1}/2 < w_i \leq \lambda_{i-1}$ , and there is a machine in the small bucket which was not used in the current phase, assign  $i$  to this machine. Finally, if all  $m$  machines in the current small bucket were used in the current phase, or if  $w_i > \lambda_{i-1}$ , then a new phase begins: we define  $\lambda_i = \max((2 - \varepsilon)\lambda_{i-1}, w_i)$  and the job is assigned to a machine in the next small bucket.

**Theorem 1.** *The algorithm *Buckets* is  $1 + \frac{1}{2^{t(1-o(1))}}$  competitive for an appropriate choice of  $\varepsilon$ .*

*Proof.* We start by showing that both invariants hold after the arrival of a job (and thus hold throughout the execution of *Buckets*). After the assignment of the first job,  $\lambda_1 = OPT_1 = w_1$ , and both invariants hold since  $\varepsilon < 1$ .

The first invariant always holds, since when a job which is larger than  $\lambda$  arrives,  $\lambda$  is modified. To show that the second invariant holds, we show that  $\lambda$

is increased only when the previous  $\lambda$  is smaller than the current  $OPT$ , and that  $\lambda$  is not increased too much. If  $\lambda$  is increased since  $\lambda_{i-1} < w_i$ , then  $OPT_i \geq w_i$  and since  $\lambda_i = \max((2 - \varepsilon)\lambda_{i-1}, w_i)$  then  $\lambda_i \leq (2 - \varepsilon)w_i \leq (2 - \varepsilon)OPT_i$ . If  $\lambda$  is increased since all the machines in the small bucket were used in the current phase, then there are at least  $m + 1$  jobs of weight more than  $\frac{\lambda_{i-1}}{2}$  and hence the optimal schedule has to assign two of them on one machine, yielding  $OPT_i > \lambda_{i-1}$ . Thus  $\lambda_i \leq (2 - \varepsilon)OPT_i$ .

Next we show that the maximum load in the big bucket never exceeds  $OPT_i$  at step  $i$  (after arrival of job  $i$ ). It is easy to see that the maximum load of running greedy on  $\alpha m$  machines is at most  $\frac{OPT_i}{\alpha} + \max_{j \leq i} w_j$ . Since  $w_j \leq \frac{\lambda_{i-1}}{2}$  and  $\lambda_{i-1}/(2 - \varepsilon) \leq OPT_{i-1}$ , the load is bounded by  $(\frac{1}{\alpha} + \frac{2-\varepsilon}{2})OPT_{i-1} \leq (\frac{\varepsilon}{2} + \frac{2-\varepsilon}{2})OPT_i = OPT_i$ .

Last, we bound the maximum load on the small bucket machines. When a new phase starts, the value of  $\lambda$  is multiplied by at least  $2 - \varepsilon$ . Each machine in a small bucket is used at most once in each phase.

Consider a job which is assigned to a small bucket machine in the last time it is used. Denote this job by  $i'$ , and let  $\lambda' = \lambda_{i'}$ . Then the previous job assigned to the same machine is of weight at most  $\lambda'/(2 - \varepsilon)^k$ . Moreover, a job that was assigned  $r \geq 1$  jobs before  $i'$  to the same machine is of weight at most  $\lambda'/(2 - \varepsilon)^{rk}$ . Thus the total weight of all jobs on this machine, except  $i'$ , is at most  $2\lambda'/(2 - \varepsilon)^k$ . Since  $OPT \geq \frac{1}{(2-\varepsilon)}\lambda'$  we get that the total weight of jobs on this machine is at most

$$w(j') + \frac{4OPT}{(2 - \varepsilon)^k} \leq (1 + \frac{4}{(2 - \varepsilon)^k})OPT \leq (1 + \frac{4}{(2 - \varepsilon)^{t-2/\varepsilon-1}})OPT.$$

Choosing an appropriate value of  $\varepsilon$  would give the required competitive ratio (for example  $\varepsilon = \sqrt{3/t}$  is a suitable value).  $\square$

## 2.2 Lower Bounds

We begin by giving a simple exponential lower bound:

**Theorem 2.** *The competitive ratio of any deterministic on-line algorithm is at least  $1 + 1/2^{2t-1}$ .*

*Proof.* We give a proof for even  $m$  and for integer  $t$ . It is easy to extend the proof for all cases. The sequence consists of  $n + \frac{m}{2}$  jobs that arrive in  $2t + 1$  phases. Phase 1 consists of  $\frac{m}{2}$  unit jobs, and phase  $i$  for  $i > 1$  consists of  $\frac{m}{2}$  jobs of weight  $2^{i-2}$ . The sequence stops after a phase in which the on-line schedules two jobs on one machine. (If the algorithm reaches the last phase, there are more jobs than on-line machines, therefore the on-line has two jobs on one machine). The optimal off-line load after every phase is the weight of the last job. If the on-line has two jobs on one machine, its load is at least  $1 + x$  where  $x$  is the weight of the last job. The minimum value of  $\frac{1+x}{x}$  would be  $1 + \frac{1}{2^{i-2}}$  where  $i = 2t + 1$ , hence  $1 + 1/2^{2t-1}$  is a lower bound on the competitive ratio.  $\square$

We can give a slightly better lower bound, this bound holds for deterministic and randomized algorithms. In fact, we show a lower bound on preemptive algorithms versus a non-preemptive optimal algorithm. Hence our lower bound holds both for the preemptive and non-preemptive models. The lower bound builds on the lower bounds given by SGALL [19] and independently by CHEN, VAN VLIET and WOEGINGER [67].

The main idea here is to use small jobs and a sequence of  $n$  big jobs  $J_i$  for  $1 \leq i \leq n$  of increasing weight so that the optimal off-line load after job  $J_i$ , which we denote by  $OPT_i$ , is exactly equal to the weight of  $J_i$ . Hence, the weight of each big job is equal to the total weight of all previous jobs divided by  $m - 1$ . Specifically, the sequence begins by very small jobs of total weight  $m - 1$  followed by the sequence of the  $n$  big jobs. The weight of  $J_i$  for  $1 \leq i \leq n$  is  $\mu^{i-1}$  where  $\mu = \frac{m}{m-1}$ .

**Lemma 2.** *The optimal off-line load for the above sequence is  $\mu^{k-1}$  after the arrival of the job  $J_k$ , for  $1 \leq k \leq n$ .*

*Proof.* We consider an algorithm which assigns all jobs on off-line machines, and show that the resulting load is  $\mu^{k-1}$ .

The algorithm assigns jobs to the off-line machines greedily, in non-increasing order (sorted according to weight). This is equivalent to using the LPT rule. We show that no big job is assigned in a way that some load exceeds  $\mu^{k-1}$ . Note that the total weight of all small jobs and first  $j$  big jobs is  $\mu^j(m-1) = \mu^{j-1}m$ .

Assume that the assignment of job  $j$  causes the maximum load to exceed  $\mu^{k-1}$ . This means that all other machines are loaded by more than  $\mu^{k-1} - \mu^{j-1}$ . Since the total weight of jobs smaller or equal to  $J_j$  is  $\mu^{j-1}m$ , we get that the total weight of jobs is more than  $\mu^{k-1}m$  which is a contradiction. Hence, the assignment of the small job results in balanced machines, each with load of  $\mu^{k-1}$ .  $\square$

The following lemma, adapted from [19, 9], is the key of lower bounding the competitive ratio.

**Lemma 3.** *For any deterministic or randomized, preemptive or non preemptive algorithms for the sequence above the following holds:  $r \geq \frac{W}{\sum_{i=1}^n OPT_i}$ , where  $r$  is the competitive ratio and  $W$  is the total weight of the jobs.*

*Proof.* Denote by  $A(J_i)$  the maximum load of the on-line algorithm  $A$  after the assignment of the job  $J_i$ . Then

$$\frac{\sum_{i=1}^n E(A(J_i))}{\sum_{i=1}^n OPT_i} \leq \frac{\sum_{i=1}^n r \cdot OPT_i}{\sum_{i=1}^n OPT_i} = r.$$

Hence it is enough to show that  $\sum_{i=1}^n E(A(J_i)) \geq W$ .

Assume that  $A$  is deterministic. For  $1 \leq l \leq n$  let  $T_l$  be the load on the  $l$ 'th machine at the end of the sequence after sorting the machines by non-increasing

load. Removing any  $l - 1$  jobs still leaves a machine with load at least  $T_l$  and thus  $A(J_l) \geq T_{n-l+1}$ . Since  $W = \sum_{k=1}^n T_k$  we conclude that

$$\sum_{i=1}^n A(J_i) \geq \sum_{i=1}^n T_{n-l+1} = W$$

as needed. If  $A$  is randomized, we average over the deterministic algorithms and conclude that

$$\sum_{i=1}^n E(A(J_i)) \geq W .$$

□

**Theorem 3.** *The competitive ratio of an on-line algorithm, deterministic or randomized, preemptive or non-preemptive, is at least  $1/(1 - (\frac{m-1}{m})^n) = 1 + 1/e^{\frac{n}{m}(1+o(1))}$ .*

*Proof.* We use the above job sequence and apply Lemma 3. We have

$$W = \mu^n(m - 1) ,$$

$$\sum_{i=1}^n OPT_i = \sum_{i=1}^n \mu^{i-1} = \frac{\mu^n - 1}{\mu - 1}$$

and

$$r \geq \frac{\mu^n(m - 1)}{(\mu^n - 1)}(\mu - 1) = \frac{\mu^n}{\mu^n - 1} = \frac{1}{1 - \frac{1}{\mu^n}} = \frac{1}{1 - (\frac{m-1}{m})^n}$$

as needed. □

We can improve the bound for the special case  $t = 2$  for the non-preemptive deterministic case.

*Claim.* The competitive ratio of any on-line algorithm for  $n = 2m$ , where  $m \geq 8$ , is at least  $\frac{5}{4}$ .

*Proof.* We use a job sequence consisting of four phases:

- $m$  jobs of weight 1
- $\lfloor \frac{m}{2} \rfloor$  jobs of weight  $3/2$
- $\lfloor \frac{m}{3} \rfloor + 1$  jobs of weight 3
- $\lfloor \frac{m+1}{6} \rfloor + 1$  jobs of weight 4.

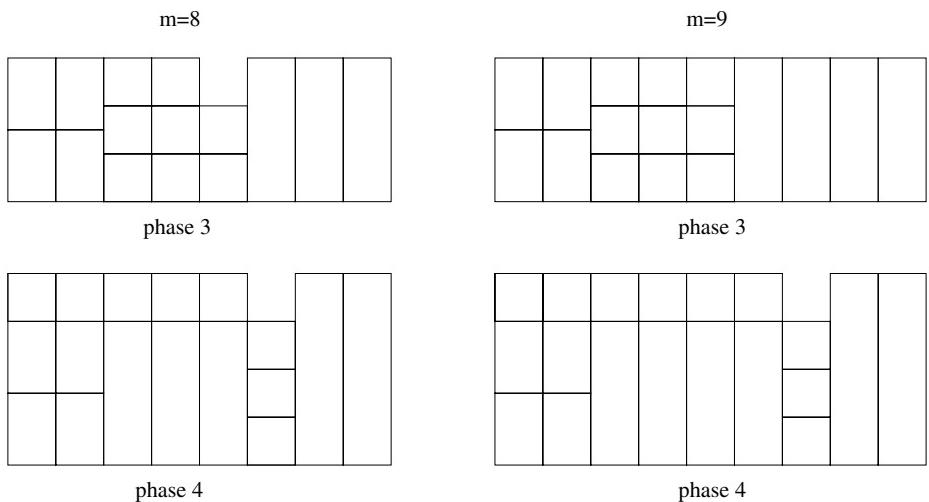
The sequence stops after a phase in which the on-line schedules two jobs on one machine. Note that the sequence contains more than  $2m$  jobs.

$m \bmod 6$	0	1	2	3	4	5
Amount of jobs	$2m + 2$	$2m + 1$				

We show that the optimal load in phase  $i$  is  $i$ . This is clear for phases 1 and 2. In phase 3, if the machines are packed to a maximum load of 3, at most 2.5 of space can be lost: 2 if a job of weight 1 has to go on its own machine, and 0.5 if there is an odd number of jobs of weight 1.5. The total weight is at most  $m + \frac{3m}{4} + (m+3) = \frac{11m}{4} + 3$ , which is at most  $3m - 2.5$  for  $m \geq 22$ . This implies that the machines can be packed with a maximum load of 3 for  $m \geq 22$ . By inspection, the machines can be packed for  $8 \leq m \leq 21$  too.

In phase 4, the total weight is at most  $\frac{11m}{4} + 3 + \frac{4m}{6} + \frac{14}{3}$ . In the optimal packing, at most 3.5 of space is lost. We have  $\frac{41}{12}m + \frac{23}{3} \leq 4m - 3.5$  which holds for  $m \geq 20$ . Therefore the optimal algorithm can maintain a load of 4 in phase 4, if  $m \geq 20$ . By inspection, it works for  $8 \leq m \leq 19$  as well.

As an example, we give the optimal schedules for phases 3 and 4 when  $m = 8$  and  $m = 9$  (see Figure 1).



**Fig. 1.** The last phases for  $m = 8, 9$

Depending on the phase in which the on-line algorithm puts two jobs on the same machine, we find competitive ratios of  $2$ ,  $\frac{5}{4}$ ,  $\frac{4}{3}$  and  $\frac{5}{4}$ . Hence the competitive ratio is at least  $5/4$ .  $\square$

### 2.3 An Optimal Preemptive Algorithm

The last part of this section presents an optimal preemptive on-line algorithm. The algorithm is similar to the algorithm in [7].

Let  $r = 1/(1 - \frac{1}{\mu^n})$ . We denote the load on machine  $i$  at time  $T$  by  $L_i^T$ . The algorithm maintains three invariants, which hold at any step  $T$ :

- $L_1^T \leq L_2^T \leq \dots \leq L_n^T$ .

- $L_n^T \leq r \cdot OPT^T$ .
- For  $1 \leq k \leq n$ ,

$$\sum_{i=1}^k L_i^T \leq \frac{\mu^k - 1}{\mu^n - 1} W^T,$$

where  $W^T$  is the total weight of jobs which arrived till time  $T$ .

Similarly to the algorithm in [7], we try to maintain a ratio of  $\frac{m}{m-1}$  between machine loads. We show how to assign a new job  $j$  with weight  $w_j$ , arriving at time  $T+1$ , to  $n$  machines. First the new optimal load is computed by  $\max(W^{T+1}/m, \max_{1 \leq i \leq T+1} w_i)$  [17], and then the following intervals are reserved for  $j$ : for  $1 \leq l \leq n-1$ , we reserve  $[L_l^T, L_{l+1}^T]$ , and for  $l = n$ , reserve  $[L_n^T, r \cdot OPT^T]$ . Note that these intervals are disjoint. Next, for  $j = n$  down to 1, assign a portion out of  $w_j$  of size equal to the size of the reserved interval. We do that until we run out of  $w_j$ . (The last portion assigned might be smaller than the interval.)

It is easy to follow the proof in [7], replacing the number of machines used by the on-line algorithm from  $m$  to  $n$ . The proof shows that each job is completely distributed to the machines and that the invariants hold. By that we conclude that the algorithm is  $r$ -competitive as required.

### 3 Temporary Tasks

Recall that for  $n = m$  the greedy algorithm is  $(2 - 1/m)$ -competitive for permanent tasks as well as for temporary tasks. Greedy is not optimal for permanent tasks, but is optimal for temporary tasks. Also for  $n > m$ , it is easy to see that greedy has the same competitive ratio for temporary tasks as for permanent tasks, which is  $1 + (m-1)/n$ . However, in contrast to the case  $n = m$ , greedy is not optimal for temporary tasks, since algorithm Buckets (defined on temporary tasks) achieves a better competitive ratio for large  $n$ . Specifically, it is easy to see that the same analysis of the competitive ratio of algorithm Buckets for permanent tasks also holds for temporary tasks. However, we show that if the online algorithm has one more machine than the optimal offline algorithm then the greedy is still optimal.

**Theorem 4.** *Greedy is optimal for temporary tasks for  $n = m + 1$ .*

**Proof.** We need to show a lower bound of  $\frac{2m}{m+1}$  on the competitive ratio of any on-line algorithm. The proof consists of two parts: one for odd  $m$  and one for even  $m$ . In the proof we mention the value of the optimal load only when the value increases.

*Case A.  $m$  is odd.* We start the sequence by  $(m-1)m^2$  unit-weight jobs. The optimal load is  $m(m-1)$ . We distinguish between two cases:

*Case A1.* The online algorithm places at least  $m(m - 1)$  jobs on one machine, say machine  $x$ .

In this case, all the jobs leave except  $m(m - 1)$  jobs on  $x$ . Then,  $m(m - 1)$  jobs of weight  $m - 1$  arrive. Since the optimal load is again  $m(m - 1)$ , at most  $m - 2$  of them can go on  $x$ . Otherwise the load would be  $(2m - 1)(m - 1)$  on  $x$ , and  $(2m - 1)/m > 2m/(m + 1)$ . So  $(m - 1)^2 + 1$  of these jobs must go on the  $m$  empty machines. We distinguish between two sub-cases:

*Case A1a.* One machine (not  $x$ ) has at least  $m$  jobs of weight  $m - 1$ .

All jobs of weight  $m - 1$  leave except  $m$  job of weight  $m - 1$  on one machine, and  $m - 1$  jobs of weight  $m(m - 1)$  arrive. The new optimal load is  $(m + 1)(m - 1)$ . Therefore all these jobs must go on different machines. Finally, a job of weight  $m(m + 1)$  arrives. This completes the proof since the online load is  $2m^2$ , while the optimal load is  $m(m + 1)$ : the last job has its own machine, the other machines have one job of weight  $m(m - 1)$ , one or two jobs of weight  $m - 1$  and some jobs of weight 1, so that the load is precisely  $m(m + 1)$ .

*Case A1b.* All machines (except machine  $x$ ) have at least one job of weight  $m - 1$ .

All jobs of weight  $m - 1$  leave except  $m$  jobs, one such job is on each machine except machine  $x$ . Next,  $\frac{m^2 - 2m - 1}{2}$  jobs of weight  $2(m - 1)$  arrive. The optimal load is again  $m(m - 1)$ . At most  $\frac{m-3}{2}$  are assigned to machine  $x$ , otherwise the load there is too large. There are  $\frac{m-3}{2} + \frac{1}{m}$  jobs on average on the other machines, so there is at least one machine (not  $x$ ) with at least  $\frac{m-1}{2}$  jobs of this weight and a load of at least  $m(m - 1)$ , say machine  $y$ . All jobs leave except the unit jobs on  $x$  and jobs of total weight precisely  $m(m - 1)$  on machine  $y$ .

Finally,  $m - 1$  jobs of weight  $m(m - 1)$  arrive and one job of weight  $m(m + 1)$ . Clearly, the online algorithm must assign each job of weight  $m(m - 1)$  to an empty machine and hence its final load is  $2m^2$ . The optimal algorithm can balance its jobs to a load of  $m(m + 1)$  since there are at least  $2(m - 1)$  jobs of weight 1, which completes the proof.

*Case A2.* All machines now have load at least  $m - 1$ .

All jobs leave except  $m - 1$  jobs on each machine, and  $m^2 - m - 1$  jobs of weight  $m - 1$  arrive. The average number of jobs of weight  $m - 1$  on the machines is  $m - 2 + \frac{1}{m+1}$ , and hence there is a machine with  $m - 1$  jobs of weight  $m - 1$  and a load of  $m(m - 1)$ . The loads are now the same as in Case A1b just before the arrival of the jobs of weight  $2(m - 1)$ . Hence, we can continue as in that case.

*Case B.*  $m$  is even. We start the sequence by  $(m - 1)m^2$  unit jobs. The optimal load is  $m(m - 1)$ . We distinguish between two cases:

*Case B1.* One machine, say  $x$ , has at least  $m(m - 1)$  jobs. All jobs leave except  $m(m - 1)$  jobs on  $x$ , and  $(m - 1)^2$  jobs of weight  $m$  arrive. The optimal load is again  $m(m - 1)$ . We distinguish between two sub-cases:

*Case B1a.* Another machine (not  $x$ ) has load at least  $m(m - 1)$ . Then all jobs of weight  $m$  leave except  $m - 1$  jobs on one machine, and  $m - 1$  jobs of weight  $m(m - 1)$  arrive followed by a job of weight  $m(m + 1)$ . Clearly, the online load is  $2m^2$ , while the optimal load is  $m(m + 1)$  which completes the proof.

*Case B1b.* Each machine except  $x$  has one job of weight  $m$ . All jobs of weight  $m$  leave except  $m$  jobs, one on each machine except on machine  $x$ . Next  $\frac{m^2 - 3m}{2}$  jobs of weight  $2m$  arrive. At most  $\frac{m-2}{2}$  can go on machine  $x$ . Hence, the average number of jobs of weight  $2m$  on machines different than  $x$  is  $\frac{m}{2} - 2 + \frac{1}{m}$ . Thus, one machine must have  $\frac{m}{2} - 1$  jobs of weight  $2m$  and a load of at least  $m(m - 1)$ . All jobs leave except the unit jobs on  $x$  and jobs of total weight  $m(m - 1)$  on the other machine. Finally,  $m - 1$  jobs of weight  $m(m - 1)$  arrive and one job of weight  $m(m + 1)$ . Clearly, the online load is  $2m^2$ , while the optimal load is  $m(m + 1)$  which completes the proof.

*Case B2.* There are at least  $m$  jobs on each machine.

All jobs leave except  $m$  jobs on each machine. Next,  $\frac{m^2(m-2)-m}{2}$  jobs of weight  $2$  arrive. If there is a machine with load at least  $m(m - 1)$ , we continue as in Case B1. Otherwise, each machine has load at least  $2m$ . Then, some jobs of weight  $2$  leave in such a way that the load on each machine is  $2(m - 1)$ . Next,  $m^2 - 2m - 2$  jobs of weight  $m - 1$  arrive. Then, one machine will have a load of at least  $m(m - 1)$ . Jobs of weight  $m - 1$  on that machine leave such that the load becomes  $m(m - 1)$ . All non-unit jobs on the other machines leave. We continue as in Case B1b.  $\square$

## 4 Conclusions

We have examined the effects of resource augmentation for several load balancing problems. For the problem of scheduling jobs on identical machine, we have shown an algorithm with a competitive ratio which decreases exponentially in  $n/m$ , while greedy has a competitive ratio that is linear in  $n/m$ .

An open question is whether it is possible to close the gap between the lower bound and the upper bound on identical machines. Both bounds are decreasing exponentially, and we conjecture that the true value of the competitive ratio is closer to the lower bound.

## Acknowledgements

The authors wish to thank Han La Poutré for helpful discussions.

## References

1. S. Albers. Better bounds for on-line scheduling. In *Proc. 29th ACM Symp. on Theory of Computing*, pages 130–139, 1997.

2. Y. Azar and L. Epstein. On-line load balancing of temporary tasks on identical machines. In *5th Israeli Symp. on Theory of Computing and Systems*, pages 119–125, 1997.
3. Y. Bartal, A. Fiat, H. Karloff, and R. Vohra. New algorithms for an ancient scheduling problem. In *Proc. 24th ACM Symposium on Theory of Algorithms*, pages 51–58, 1992. To appear in *Journal of Computer and System Sciences*.
4. P. Berman and C. Coulston. Speed is more powerful than clairvoyance. In *Nordic Journal of Computing*, pages 181–193, 1999.
5. M. Brehob, E. Torng, and P. Uthaisombut. Applying extra-resource analysis to load balancing. Manuscript, 1999.
6. B. Chen, A. van Vliet, and G. J. Woeginger. Lower bounds for randomized online scheduling. *Information Processing Letters*, 51:219–222, 1994.
7. B. Chen, A. van Vliet, and G. J. Woeginger. An optimal algorithm for preemptive on-line scheduling. *Operations Research Letters*, 18:127–131, 1995.
8. J. Edmonds. Scheduling in the dark. In *Proceedings of the 31st ACM Symposium on Theory of Computing*, pages 179–188, 1999.
9. L. Epstein and J. Sgall. A lower bound for on-line scheduling on uniformly related machines. *To appear in Oper. Res. Lett.*, 2000.
10. T. Gormley, N. Reingold, E. Torng, and J. Westbrook. Generating adversaries for request-answer games. In *Proc. 11th ACM-SIAM Symp. on Discrete Algorithms*, 2000.
11. R.L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.
12. R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17:263–269, 1969.
13. B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. In *Proceedings of 36th IEEE Symposium on Foundations of Computer Science*, pages 214–221, 1995.
14. Bala Kalyanasundaram and Kirk Pruhs. Maximizing job completions online. In *European Symposium on Algorithms*, pages 235–246, 1998.
15. D. R. Karger, S. J. Phillips, and E. Torng. A better algorithm for an ancient scheduling problem. In *Proc. of the 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 132–140, 1994.
16. Tak Wah Lam and Kar Keung To. Trade-offs between speed and processor in hard-deadline scheduling. In *ACM/SIAM Symposium on Discrete Algorithms*, pages 623–632, 1999.
17. R. McNaughton. Scheduling with deadlines and loss functions. *Management Sci.*, 6:1–12, 1959.
18. Cynthia A. Philips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 140–149, 1997.
19. J. Sgall. A lower bound for randomized on-line multiprocessor scheduling. *Inf. Process. Lett.*, 63(1):51–55, 1997.
20. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

# Fair versus Unrestricted Bin Packing

Yossi Azar<sup>1,\*</sup>, Joan Boyar<sup>2,\*\*</sup>, Lene M. Favrholdt<sup>2</sup>,  
Kim S. Larsen<sup>2,\*\*</sup>, and Morten N. Nielsen<sup>2</sup>

<sup>1</sup> Department of Computer Science, Tel-Aviv University, [azar@math.tau.ac.il](mailto:azar@math.tau.ac.il)

<sup>2</sup> Department of Mathematics and Computer Science, University of Southern Denmark, Odense, [{joan, lenem, kslarsen, nyhave}@imada.sdu.dk](mailto:{joan, lenem, kslarsen, nyhave}@imada.sdu.dk).

**Abstract.** We consider the Unrestricted Bin Packing problem where we have bins of equal size and a sequence of items. The goal is to maximize the number of items that are packed in the bins by an on-line algorithm. We investigate the power of performing admission control on the items, i.e., rejecting items while there is enough space to pack them, versus behaving fairly, i.e., rejecting an item only when there is not enough space to pack it. We show that by performing admission control on the items, we get better performance for various measures compared with the performance achieved on the fair version of the problem. Our main result shows that we can pack  $2/3$  of the items for sequences in which the optimal can pack all the items.

## 1 Introduction

### 1.1 General

In this paper, we are investigating the competitive ratio for a bin packing problem. However, in addition to considering unrestricted request sequences, we also consider some restricted sequences which we refer to as accommodating sequences. Informally, these are sequences where an optimal algorithm can satisfy all requests. Clearly, the competitive ratio on accommodating sequences<sup>1</sup> is no worse than the competitive ratio on unrestricted sequences for any given problem and sometimes can be much better. For problems where the competitive ratio is a bad measure, it may be useful to compare algorithms by their competitive ratio on accommodating sequences. Specifically, it was shown in [4,5] that there are (benefit) problems where the competitive ratio tends to zero while the competitive ratio on accommodating sequences is a constant, i.e., independent of the parameters of the problem. Moreover, when we are trying to distinguish between two algorithms, the competitive ratio on accommodating sequences may prefer one algorithm while the competitive ratio measure (on all sequences) prefers the other [5].

\* Supported in part by the Israel Science Foundation, and by a USA-Israel BSF grant.

\*\* Supported in part by the Danish Natural Science Research Council (SNF).

<sup>1</sup> In earlier papers [4,5,6], this competitive ratio on accommodating sequences was called the accommodating ratio. The change is made here for consistency with common practice in the field.

In the *Bin Packing problem* we are given some bins and the goal is to pack a set of items into these bins. We concentrate on the benefit variant of the problem, where there are  $n$  bins and the objective is to maximize the total number of items in these bins. This problem has been studied in the off-line setting, starting in [8], and its applicability to processor and storage allocation is discussed in [9]. (For surveys on bin packing, see [107].)

In the on-line version of the problem the items arrive in some sequence and the assignment of an item should be done before the next item arrives. We assume that the items are integer-sized and the bins all have size  $k$ . One can discuss the Fair Bin Packing problem<sup>2</sup> where it is required that the packing be *fair*, that is, an item can only be rejected if it cannot fit in any bin at the time when it is given. Note that the optimal algorithm is also required to be fair. It is shown in [5] that for this problem, Worst-Fit has a strictly better competitive ratio than First-Fit, while First-Fit has a strictly better competitive ratio than Worst-Fit on accommodating sequences. In this case, the competitive ratio on accommodating sequences seems the more appropriate measure, since it is constant while the competitive ratio (on all sequences) is close to zero, for large values of  $k$ , basically due to some sequences which seem very contrived. This demonstrated the usefulness of the more general accommodating function [6] which comprises the competitive ratio as well as the competitive ratio on accommodating sequences (it is a function of the restriction on the request sequences).

Here, we consider what happens when the fairness restriction is removed. Thus, for the on-line problem *Unrestricted Bin Packing (UBP)*, there are again  $n$  bins, all of size  $k$ , the items are integer-sized, and the goal is to maximize the total number of items placed in the bins, but there is no fairness restriction.

We note that on accommodating sequences, the competitive ratio of UBP is no worse than the competitive ratio of the fair problem, since the optimal algorithm serves all the requests and hence is fair. In general, however, the competitive ratio of UBP is not necessarily better than the competitive ratio of the fair problem since the optimal algorithms may be different. In fact, in many cases, considering unfair algorithms, i.e., performing admission control on the requests, is the more challenging problem; see for example the results for throughput routing in [123]. In particular, with the Unrestricted Bin Packing problem, it is easier to differentiate between algorithms since both their competitive ratio and their competitive ratio on accommodating sequences can vary over a large range. This is in contrast to on-line algorithms for Fair Bin Packing where all of them must have both within a constant factor of each other.

## 1.2 Accommodating Sequences and the Accommodating Function

For completeness, we define the competitive ratio and the accommodating function for Unrestricted Bin Packing. Note that Unrestricted Bin Packing is a maximization problem, and all ratios are less than or equal to 1.

---

<sup>2</sup> In [6] where some of the results from [5] were first presented in a preliminary form, this problem was called Unit Price Bin Packing.

Let  $\mathbb{A}(I)$  denote the number of items algorithm  $\mathbb{A}$  accepts when given request sequence  $I$  and let  $\text{OPT}(I)$  denote the number an optimal off-line algorithm,  $\text{OPT}$ , accepts. An on-line algorithm,  $\mathbb{A}$ , is  $c$ -competitive if there exists a constant  $b$ , such that  $\mathbb{A}(I) \geq c \cdot \text{OPT}(I) - b$  for all sequences  $I$ . The competitive ratio  $CR = \sup\{c \mid \mathbb{A} \text{ is } c\text{-competitive}\}$ .

Next, we introduce the restricted request sequences. We say that  $I$  is an  $\alpha$ -sequence, if  $I$  could be packed in  $\alpha n$  bins. We investigate the competitive ratio on such restricted sequences. To be precise, an on-line algorithm  $\mathbb{A}$  is  $c$ -competitive on  $\alpha$ -sequences if  $c \leq 1$  and there exists a constant  $b$ , such that for every  $\alpha$ -sequence  $I$ ,  $\mathbb{A}(I) \geq c \cdot \text{OPT}(I) - b$ . The accommodating function  $\mathcal{A}$  is defined as  $\mathcal{A}(\alpha) = \sup\{c \mid \mathbb{A} \text{ is } c\text{-competitive on } \alpha\text{-sequences}\}$ .

Thus, the accommodating function for an algorithm is the competitive ratio of that algorithm on  $\alpha$ -sequences as a function of  $\alpha$ . We refer to 1-sequences as *accommodating sequences*, since the optimal algorithm can accommodate all requests in such a sequence. We use  $AR$  to denote the competitive ratio on accommodating sequences.

### 1.3 Results

We prove results on the Unrestricted Bin Packing problem for the usual competitive ratio, the competitive ratio on accommodating sequences and the accommodating function. We start with the competitive ratio.

For the usual competitive ratio we prove the following:

- The algorithm Log (Section 2.2) has a competitive ratio of  $\Theta(\frac{1}{\log k})$ .
- No on-line algorithm can have a competitive ratio which is better than  $O(\frac{1}{\log k})$ , even when considering randomized algorithms.
- We observe that the competitive ratios of First-Fit and Worst-Fit are  $\frac{1}{k}$ .

These results should be compared with the competitive ratio of any on-line algorithm for the fair problem: they are all  $\Theta(\frac{1}{k})$  [5].

For the competitive ratio on accommodating sequences we prove:

- The competitive ratio of Log on accommodating sequences is  $\Theta(\frac{1}{\log k})$ .
- We conclude from [5] that the competitive ratio of First-Fit on accommodating sequences is between  $\frac{5}{8}$  and  $\frac{7}{11}$ , since the fairness restriction on  $\text{OPT}$  is irrelevant when all of the items can be packed.
- We design an unrestricted algorithm, Unfair-First-Fit, whose competitive ratio on accommodating sequences is  $\frac{2}{3}$ , which is strictly higher than the competitive ratio of First-Fit on accommodating sequences.
- The competitive ratio of any on-line algorithm on accommodating sequences is no better than  $\frac{6}{7}$ , even when considering randomized algorithms.

Thus, according to the usual competitive ratio, Log is the better algorithm, and according to the competitive ratio on accommodating sequences, First-Fit is the better algorithm (the same is true for Log and Unfair-First-Fit).

For the accommodating function we prove the following:

- We design randomized and deterministic algorithms for which the accommodating function evaluated at any constant  $\alpha$  is a constant, if the algorithm is given the value  $\alpha$ .
- In contrast, we observe that First-Fit's (and Unfair-First-Fit's) accommodating function drops down to  $\Theta(\frac{1}{k})$  for  $\alpha \geq 1 + c$ , for any constant  $c > 0$ .

The main technical effort is to prove the competitive ratio of the algorithm Unfair-First-Fit on accommodating sequences. The other results are easier to prove. Algorithm Log uses derandomization of the standard classify and select technique. The proof of the lower bound for Log is similar to the lower bound proof in [2], and the proof of the general upper bound for the competitive ratio is analogous to the proof of the corresponding lemma in [1].

*Remark:* In this paper, we assume that all items are integer-sized and the bins have size  $k$ . All of the results hold with the weaker assumption that the bins are unit-sized and the smallest item has size at least  $\frac{1}{k}$ . However, some of the results in [5] do not appear to hold with this assumption, so we use the stronger assumption for consistency.

## 2 The Competitive Ratio

### 2.1 First-Fit and Worst-Fit

It is easy to see that the competitive ratio of First-Fit or Worst-Fit for Unrestricted Bin Packing is  $\frac{1}{k}$ . For the upper bound, consider the sequence consisting of  $n$  items of size  $k$  followed by  $n \cdot k$  items of size 1. For the lower bound, note that if First-Fit (or Worst-Fit) rejects anything, it accepts at least  $n$  items, and no algorithm can accept more than  $n \cdot k$  items. From that it follows that First-Fit's (and Worst-Fit's) accommodating function drops down to  $\frac{1}{k}$  for  $\alpha \geq 2$ . Moreover, it is  $\Theta(\frac{1}{k})$  for  $\alpha \geq 1 + c$ , for any constant  $c > 0$ , by using  $(\alpha - 1)n \cdot k$  (instead of  $n \cdot k$ ) items of size 1.

### 2.2 Algorithm Log

In the description of the algorithm Log, we assume that  $n > c \lceil \log_2 k \rceil$ , for some constant  $c > 1$ . If  $n$  is smaller, we can use simple randomization to achieve the same results.

Log divides the  $n$  bins into  $\lceil \log_2 k \rceil$  groups  $G_1, G_2, \dots, G_{\lceil \log_2 k \rceil}$ . Let  $p = \lfloor \frac{n}{\lceil \log_2 k \rceil} \rfloor$  and let  $s = n - p \cdot \lceil \log_2 k \rceil$ . Groups  $G_1, G_2, \dots, G_s$  consist of  $p + 1$  bins and the rest of the groups consist of  $p$  bins. Let  $S_1 = \{x \mid \frac{k}{2} \leq x \leq k\}$ , and  $S_i = \{x \mid \frac{k}{2^i} \leq x < \frac{k}{2^{i-1}}\}$ , for  $2 \leq i \leq \lceil \log_2 k \rceil$ . When Log receives an item  $o$  of size  $s_o \in S_i$ , it decides which group  $G_j$  of bins to pack it in by calculating  $j = \max\{j \leq i \mid \text{there is a bin in } G_j \text{ that has room for } o\}$ . If  $j$  exists,  $o$  is packed in  $G_j$  according to the First-Fit packing rule. If not, the item  $o$  is rejected.

**Theorem 1.** *The competitive ratio of Log is  $\Theta(\frac{1}{\log k})$ , even on accommodating sequences.*

*Proof.* Consider first the lower bound. For  $i \in \{1, 2, \dots, \lceil \log_2 k \rceil\}$ , let  $n_i(I)$  denote the number of items of size  $s \in S_i$  accepted by OPT when given the sequence  $I$  of items. Since group  $G_i$  is reserved for items of size  $\frac{k}{2^{i-1}}$  or smaller, the bins in group  $G_i$  will receive at least  $\min\{2^{i-1}p, n_i(I)\}$  items. OPT can accept at most  $2^i n$  items with sizes in  $S_i$ , i.e.  $n_i(I) \leq 2^i n$ . Thus,  $2^{i-1}p > 2^{i-1}(\frac{n}{\lceil \log_2 k \rceil} - 1) \geq n_i(I)(\frac{1}{2\lceil \log_2 k \rceil} - \frac{1}{2n})$ . Given the same sequence, Log packs at least  $n_i(I)(\frac{1}{2\lceil \log_2 k \rceil} - \frac{1}{2n})$  items in  $G_i$ , for  $i \in \{1, 2, \dots, \lceil \log_2 k \rceil\}$ . So, for any  $I$ ,

$$\frac{\text{Log}(I)}{\text{OPT}(I)} > \frac{\sum_{i \in \{1, 2, \dots, \lceil \log_2 k \rceil\}} n_i(I)(\frac{1}{2\lceil \log_2 k \rceil} - \frac{1}{2n})}{\sum_{i \in \{1, 2, \dots, \lceil \log_2 k \rceil\}} n_i(I)} = \frac{1}{2\lceil \log_2 k \rceil} - \frac{1}{2n},$$

so  $CR_{\text{Log}} > \frac{1}{2\lceil \log_2 k \rceil} - \frac{1}{2n}$ .

For the upper bound, consider the sequence  $I$  with  $n$  items of size  $k$ . Then,

$$\frac{\text{Log}(I)}{\text{OPT}(I)} = \frac{\lceil \frac{n}{\lceil \log_2 k \rceil} \rceil}{n} < \frac{1}{\lceil \log_2 k \rceil} + \frac{1}{n},$$

so  $AR_{\text{Log}} < \frac{1}{\lceil \log_2 k \rceil} + \frac{1}{n}$ . Since all sequences are considered for the competitive ratio,  $CR_{\text{Log}} \leq AR_{\text{Log}}$ , and the result follows.  $\square$

### 2.3 An Upper Bound on the Competitive Ratio

In this section, we consider an arbitrary on-line algorithm  $\mathbb{A}$  for Unrestricted Bin Packing and prove general bounds on how well it can do. First, note that the only possible lower bound on the competitive ratio, even on accommodating sequences, is zero, since for the algorithm which simply rejects everything, the ratio is equal to zero.

Clearly, the algorithm Log does not have the best possible competitive ratio on accommodating sequences, but its competitive ratio is quite close to optimal.

**Theorem 2.** *Any deterministic or randomized algorithm for Unrestricted Bin Packing has a competitive ratio of less than  $\frac{2}{\log_2 k}$ .*

*Proof.* Assume that  $k$  is a power of 2. The items are given in phases numbered  $0, 1, \dots, r$ ,  $r \leq \log_2 k$ . In phase  $i$ ,  $n2^i$  items of size  $k/2^i$  are given. Clearly, any optimal off-line algorithm will accept all  $n2^r$  items in phase  $r$ .

Let  $x_i$  be the expected number of items that the on-line algorithm accepts in phase  $i$ ,  $0 \leq i \leq r$ , and  $x_i = 0$ ,  $r < i \leq \log_2 k$ . By the linearity of expectations, the expected total number of items accepted by the on-line algorithm is  $\sum_{i=0}^{\log_2 k} x_i$  and the expected total volume of the items accepted is  $\sum_{i=0}^{\log_2 k} k2^{-i}x_i$ . Since there are only  $nk$  units of capacity overall, we get:  $\sum_{i=0}^{\log_2 k} k2^{-i}x_i \leq nk$ , or  $\sum_{i=0}^{\log_2 k} 2^{-i}x_i \leq n$ .

We now show that  $r$  can be chosen such that  $\sum_{i=0}^r x_i < \frac{2 \cdot n \cdot 2^r}{\log_2 k}$ , meaning that OPT will pack more than  $\frac{1}{2} \log_2 k$  times as many items as the on-line algorithm. Defining  $S_j = 2^{-j} \sum_{i=0}^j x_i$ , this statement can be reformulated as  $\exists r \in \{0, 1, \dots, \log_2 k\} : S_r < \frac{2n}{\log_2 k}$ , which is proven by the following inequality.

$$\sum_{j=0}^{\log_2 k} S_j = \sum_{0 \leq i \leq j \leq \log_2 k} 2^{-j} x_i < \sum_{i=0}^{\log_2 k} 2 \cdot 2^{-i} x_i \leq 2n. \quad \square$$

### 3 The Competitive Ratio on Accommodating Sequences

#### 3.1 An Upper Bound

Now we turn to the competitive ratio on accommodating sequences. In [5], it was shown that for  $k \geq 7$ , any deterministic Fair Bin Packing algorithm has a competitive ratio on accommodating sequences of at most  $\frac{6}{7}$ . The same result and essentially the same proof hold when the fairness restriction is removed, even for randomized algorithms.

**Theorem 3.** *For  $k \geq 7$ , any deterministic or randomized Unrestricted Bin Packing algorithm has a competitive ratio of at most  $\frac{6}{7}$ , even on accommodating sequences.*

*Proof.* Assume  $n$  is even. Consider an arbitrary on-line algorithm A. An adversary can proceed as follows: Give  $n$  items of size  $\lceil \frac{k}{2} \rceil - 1$ , and let  $q$  denote the number of bins which contain two items after this. In the case where  $E[q] < \frac{2n}{7}$ , the adversary gives  $\frac{n}{2}$  long requests of size  $k$ . The off-line algorithm can pack the first  $n$  requests in the first  $\frac{n}{2}$  bins and thus accept all  $\frac{3n}{2}$  items. On average, the on-line algorithm places two items in  $E[q]$  bins and has at most one item in every other bin. The performance ratio is thus at most  $\frac{E[n+q]}{n+\frac{n}{2}} = \frac{2n+2q}{3n} < \frac{6}{7}$ .

In the case where  $E[q] \geq \frac{2n}{7}$ , the adversary gives  $n$  requests of size  $\lfloor \frac{k}{2} \rfloor + 1$ . The off-line algorithm can pack the first  $n$  items one per bin and thus accept all  $2n$  items. The on-line algorithm must reject at least  $E[q]$  items on average. The performance ratio is thus at most  $\frac{E[2n-q]}{2n} \leq \frac{6}{7}$ .  $\square$

#### 3.2 Unfair-First-Fit

**The Algorithm.** In Section 2.2, it was shown that there is an algorithm for Unrestricted Bin Packing which has a better competitive ratio than any algorithm for Fair Bin Packing. It would be difficult to do the same for the competitive ratio on accommodating sequences, since the best upper bound known is  $\frac{6}{7}$  for both problems. First-Fit's competitive ratio on accommodating sequences is known to lie between  $\frac{5}{8}$  and  $\frac{7}{11}$  [6], and no algorithm for Fair Bin Packing is known to have a better competitive ratio on accommodating sequences. The algorithm Unfair-First-Fit (UFF), presented below, is shown to have a competitive ratio on accommodating sequences which is better than that of First-Fit as long as the

number of bins is at least 22; the ratio approaches  $\frac{2}{3}$  as  $n$  increases. What makes Unfair-First-Fit different from First-Fit is that items larger than  $\frac{k}{2}$  are rejected if enough items have been accepted already to maintain the desired ratio of  $\frac{2}{3}$ .

```

Input:  $S = \langle o_1, o_2, \dots, o_n \rangle$ 
Output:  $A$ ,  $R$ , and a packing for those items in  $A$ 
 $A := \{o_1\}$ ;  $R := \{\}$ ;  $S := \text{tail}(S)$ 
while  $S \neq \langle \rangle$ 
     $o := \text{hd}(S); S := \text{tail}(S)$ 
    if  $\text{size}(o) > \frac{k}{2}$  and  $\frac{|A|}{|A|+|R|+1} \geq \frac{2}{3}$ 
         $R := R \cup \{o\}$ 
    else if there is space for  $o$  in some bin
        place  $o$  according to the First-Fit rule
         $A := A \cup \{o\}$ 
    else
         $R := R \cup \{o\}$ 

```

### The Competitive Ratio on Accommodating Sequences.

**Theorem 4.** For  $n \geq 9$ , the competitive ratio of Unfair-First-Fit on accommodating sequences is more than  $\frac{2}{3} - \frac{4}{6n+3}$ . Thus, for  $n \geq 22$ ,  $AR_{UFF} > AR_{FF}$ .

*Proof.* The term “large” is used for items strictly larger than  $\frac{k}{2}$ , since they are considered in a special way by the algorithm. Let  $B$  denote the set of large items that are alone in a bin in UFF’s packing. Let  $s$  denote the size of the smallest item in  $R$ . We divide the proof into two cases depending on the size of  $s$ . The first case is easy.

*Case 1:  $s > \frac{k}{2}$ :* Since the smallest item in  $R$  is larger than  $\frac{k}{2}$ , the items in  $R \cup B$  are all larger than  $\frac{k}{2}$ . Thus, since all items can be packed in  $n$  bins,  $|R| + |B| \leq n$ , or  $|R| \leq n - |B|$ . Furthermore, at most one small item can be alone in a bin:  $|A| \geq 2n - |B| - 1$ . Thus, the performance ratio is

$$\frac{|A|}{|A| + |R|} \geq \frac{2n - |B| - 1}{2n - |B| - 1 + n - |B|} \geq \frac{2n - 1}{3n - 1} = \frac{2}{3} - \frac{1}{9n - 3}.$$

*Case 2:  $s \leq \frac{k}{2}$ :* Since we consider the competitive ratio on accommodating sequences, an optimal off-line algorithm, OPT, can pack all items in  $S$ . It may be instructive to view the optimal packing as being done in 3 phases:

1. UFF is run on  $S$ .
2. The packed items are rearranged, creating room for the rejected items.
3. The rejected items are packed.

The packing after Phase 1 is denoted by  $P_{\text{UFF}}$ , and the packing after Phase 3 is denoted by  $P_{\text{OPT}}$ . Similarly,  $E_{\text{UFF}}$  and  $E_{\text{OPT}}$  are used to denote the total empty space after Phase 1 and Phase 3 respectively. We assume without loss of generality that no large item is moved during Phase 2.

We divide the rejected items into two disjoint sets:  $R_b$  which contains large items, and  $R_s$  which contains small items. We use the following equation to bound the number of small items rejected.

$$|R_s| \leq \frac{1}{s} \cdot \left( E_{\text{UFF}} - E_{\text{OPT}} - \frac{k}{2} |R_b| \right)$$

It is easy to see that  $|R| < n$ , since the empty space in any bin in  $P_{\text{UFF}}$  is less than  $s$  and all rejected items have size at least  $s$ . Thus, if all bins contain at least two items each,  $\frac{|A|}{|A|+|R|} > \frac{2n}{2n+n} = \frac{2}{3}$  and we are through. Therefore, assume that some bins contain only one item. Since the empty space in any bin is less than  $\frac{k}{2}$ , such items must be large. Thus, the items that are alone in a bin are exactly the items in  $B$ .

It is now clear that  $|A| \geq 2n - |B|$ . However, if some bins contain more than two items, this lower bound is too pessimistic. Therefore, we try to “spread out” the items a little more. Assume that the items in  $P_{\text{UFF}}$  are labeled with consecutive numbers in each bin according to their arrival time, i.e., the first item in a bin is labeled 1, the next one is labeled 2, and so on. We split Phase 2 into two Subphases, 2A and 2B, such that in Subphase 2A only items with labels higher than 2 are moved and in Subphase 2B the remaining moves are performed. Note that the packing produced during Subphase 2A is only technical and used for counting purposes; it might be illegal in that some bins might contain a total volume larger than  $k$ .

If some of the items moved during Subphase 2A are moved to bins containing items from  $B$ , a better lower bound on  $|A|$  can now be obtained (Lemma 1). The set of items that are still alone after Subphase 2A is divided into two sets:  $X$ , containing the items that are still alone after Subphase 2B, and  $L$ , containing those that are not. Any item that is alone after Subphase 2A was alone in  $P_{\text{UFF}}$  as well. Since no such item can be combined with an item belonging to  $R$ , each item in  $X$  is also alone in  $P_{\text{OPT}}$ . Therefore, the bins containing an item from  $X$  do not contribute to  $E_{\text{UFF}} - E_{\text{OPT}}$ .

**Lemma 1.**  $|A| \geq 2n - |L| - |X|$ .

*Proof.*  $L \cup X$  is the set of objects that are alone after Subphase 2A.  $\square$

The following easy lemma is used to prove Lemma 3 below which, loosely speaking, shows that if we cannot guarantee that most of the bins contain at least two items after Subphase 2A, then much of the empty space in  $P_{\text{UFF}}$  is used by large rejected items.

Let  $t$  denote the time just after the last large item was accepted by UFF and let  $A_t$  denote the set of items accepted at time  $t$ .

**Lemma 2.**  $|R_b| \geq \frac{1}{2}|A_t| - 1$ .

*Proof.* Since a large item was accepted just before time  $t$ , all items previously rejected are large items and therefore contained in  $R_b$ . Since the item was accepted,  $\frac{|A_t|-1}{|A_t|-1+|R_b|+1} < \frac{2}{3}$ . Solving for  $|R_b|$ , we get  $|R_b| > \frac{1}{2}|A_t| - \frac{3}{2}$ , and since  $|R_b|$  must be integer, we get  $|R_b| \geq \frac{1}{2}|A_t| - 1$ .  $\square$

Assume that at time  $t$  all small items accepted by UFF are marked.

**Lemma 3.**  $|R_b| \geq |L| + \frac{1}{2}|X| - 1$ .

*Proof.* It is shown that  $|A_t| \geq 2|L| + |X|$ , which will complete the proof, since, by Lemma 2,  $|R_b| \geq \frac{1}{2}|A_t| - 1$ . To each item  $o \in L$ , a marked item is assigned in the following way. Since no item in  $L$  is alone after Phase 2, we can assume that the bin  $b_o$  containing  $o$  will receive at least one item,  $o'$ , labeled 1 or 2 during Phase 2. If  $o'$  is marked, it is assigned to  $o$ . Otherwise, it must be labeled 2, since all items labeled 1 in bins before  $b_o$  are marked. The item which was packed below  $o'$  in  $P_{\text{UFF}}$  was alone at time  $t$ . Therefore, this item is not moved to any item in  $L$ . This item (labeled 1) can be assigned to  $o$ . In this way, every item in  $L$  has an item assigned which arrived before time  $t$  and which is not in  $L \cup X$ . Since  $L \cup X \subseteq A_t$ ,  $|A_t| \geq 2|L| + |X|$ .  $\square$

*Subcase 2a:*  $s \leq \frac{k}{3}$ . Since the smallest item in  $R$  has size  $s$ , the empty space in each bin in  $P_{\text{UFF}}$  is smaller than  $s$ . Thus, we can use  $s(n - |X|)$  as an upper bound on  $E_{\text{UFF}} - E_{\text{OPT}}$ :

$$\begin{aligned} |R_s| &\leq \frac{1}{s} \cdot \left( E_{\text{UFF}} - E_{\text{OPT}} - \frac{k}{2}|R_b| \right) < \frac{1}{s} \left( s(n - |X|) - \frac{k}{2}|R_b| \right) \\ &= n - |X| - \frac{k}{2s}|R_b| \leq n - |X| - \frac{3}{2}|R_b|. \end{aligned}$$

Now, using Lemma 3, we get

$$\begin{aligned} |R| &= |R_s| + |R_b| \leq n - |X| - \frac{1}{2}|R_b| \leq n - |X| - \frac{1}{2} \left( |L| + \frac{1}{2}|X| - 1 \right) \\ &= n - \frac{5}{4}|X| - \frac{1}{2}|L| + \frac{1}{2}. \end{aligned}$$

Thus,

$$\begin{aligned} \frac{|A|}{|A| + |R|} &\geq \frac{2n - |L| - |X|}{2n - |L| - |X| + (n - \frac{5}{4}|X| - \frac{1}{2}|L| + \frac{1}{2})} \\ &\geq \frac{2n - (|L| + |X|) + \frac{1}{3}}{3n - \frac{3}{2}(|L| + |X|) + \frac{1}{2}} - \frac{\frac{1}{3}}{3n - \frac{3}{2}(|L| + |X|) + \frac{1}{2}} \\ &\geq \frac{2}{3} - \frac{2}{12n - 3}, \end{aligned}$$

since  $|L| + |X| \leq \frac{2}{3}(n + 1)$ , which follows from the fact that the number of large items is at most  $n$ :  $n \geq |R_b| + |L| + |X| \geq (|L| + \frac{1}{2}|X| - 1) + |L| + |X| \geq \frac{3}{2}(|L| + |X|) - 1$ .

*Subcase 2b:*  $\frac{k}{3} < s \leq \frac{k}{2}$ . In this case,  $s(n - |X|)$  is not a good bound on  $E_{\text{UFF}} - E_{\text{OPT}}$ , but we will show that even in this case,  $E_{\text{UFF}} - E_{\text{OPT}}$  is “almost” bounded by  $\frac{k}{3}(n - |X|)$ , if  $n \geq 9$  and  $\frac{|A|}{|A| + |R|} < \frac{2}{3}$ . Lemma 4 below is used for this purpose.

**Lemma 4.** *Let  $m$  be the number of bins containing at least  $c$  items in a First-Fit packing. If  $c \geq 1$  and  $m \geq c + 1$ , then the volume  $V$  of the items in these  $m$  bins is more than  $\frac{c}{c+1}mk$ .*

*Proof.* Let  $C$  denote the set of bins containing at least  $c$  items, and, for any bin  $b$ , let  $V(b)$  denote the sum of the sizes of the items in  $b$ .

Suppose, for the sake of contradiction, that  $V \leq \frac{c}{c+1}mk$ . Then there is a bin  $b \in C$  such that  $V(b) = \frac{c}{c+1}k - \varepsilon$ ,  $\varepsilon \geq 0$ . The size of any item placed in a bin to the right of  $b$  must be greater than  $\frac{1}{c+1}k + \varepsilon$ , since otherwise it would fit in  $b$ . Therefore any bin  $b' \in C$  to the right of  $b$  has  $V(b') > \frac{c}{c+1}k + c\varepsilon \geq \frac{c}{c+1}k$ . This means that there is only one bin  $b \in C$  with  $V(b) \leq \frac{c}{c+1}k$ , and if  $b$  is not the rightmost nonempty bin in  $C$ , then  $V > (m-2)\frac{c}{c+1}k + (\frac{c}{c+1}k - \varepsilon) + (\frac{c}{c+1}k + c\varepsilon) \geq m\frac{c}{c+1}k$ . Thus,  $b$  must be the rightmost nonempty bin in  $C$ .

One of the items in  $b$  must have size at most  $\frac{1}{c+1}k - \frac{\varepsilon}{c}$ . Since this item was not placed in one of the  $m-1$  bins to the left of  $b$ , these must all be filled to more than  $\frac{c}{c+1}k + \frac{\varepsilon}{c}$ . Thus,  $V > (m-1)(\frac{c}{c+1}k + \frac{\varepsilon}{c}) + (\frac{c}{c+1}k - \varepsilon) = m\frac{c}{c+1}k + (m-1)\frac{\varepsilon}{c} - \varepsilon \geq m\frac{c}{c+1}k + c\frac{\varepsilon}{c} - \varepsilon = m\frac{c}{c+1}k$ , which is a contradiction.  $\square$

Assuming  $n \geq 9$ , Lemma 4 combined with Lemma 5 below says that the average empty space in bins containing more than one item can be assumed to be at most  $\frac{k}{3}$ .

**Lemma 5.** *Assume that  $n \geq 9$  and  $s \leq \frac{k}{2}$ . Then, in  $P_{\text{UFF}}$ , at least three bins contain two or more items.*

*Proof.* Assume for the sake of contradiction that fewer than three bins contain at least two items. Since  $s \leq \frac{k}{2}$ , no bin contains a single item of size at most  $\frac{k}{2}$ . Therefore, at least  $n-2$  bins contain large items, which all arrived before time  $t$ , i.e.,  $A_t \geq n-2$ . By Lemma 2, at least  $\frac{1}{2}A_t - 1$  large items are rejected. Adding these up and noting that there can be at most  $n$  large items, we get  $n-2 + \frac{n-2}{2} - 1 \leq n$ . Solving for  $n$  yields  $n \leq 8$ , which is a contradiction.  $\square$

Our goal is now, roughly speaking, to show that the average empty space in all  $n$  bins is bounded by approximately  $\frac{k}{3}$ . Number the bins from left to right, and let  $l$  be the number of the bin in which the last large item was placed. Let  $e$  denote the largest empty space in bins containing an item from  $B$ . In the proof of Lemma 7 we will show a lower bound on the number of bins to the right of  $l$  of approximately  $\frac{|B|}{2}$ . Each of these bins contains at least two items of size larger than  $e$ . Thus, even if  $e > \frac{k}{3}$ , the average empty space in the  $B$ -bins and the bins to the right of  $l$  will be bounded above by approximately  $(|B|e + (k-2e)\frac{|B|}{2}) / \frac{3|B|}{2} = \frac{k|B|}{2} \cdot \frac{2}{3|B|} = \frac{k}{3}$ . Lemma 4 combined with Lemma 6 below says that we can assume that the rest of the bins have an average empty space of at most  $\frac{k}{3}$ .

**Lemma 6.** Assume that  $n \geq 9$ ,  $s \leq \frac{k}{2}$ ,  $e \geq \frac{k}{3}$ , and  $\frac{|A|}{|A|+|R|} < \frac{2}{3}$ . Then, in  $P_{UFF}$  at least three of the first  $l$  bins contain two or more items.

*Proof.* We count the total number of items of size larger than  $e$ . Since  $|A| \geq 2n - |B|$ , more than  $n - \frac{|B|}{2}$  items are rejected, because otherwise we have a performance ratio of  $\frac{2}{3}$ , which is a contradiction. After bin  $l$ , there are  $n - l$  bins containing at least two items each. All of the rejected items and those in the last  $n - l$  bins are larger than  $e$  and there are more than  $n - \frac{|B|}{2} + 2(n - l)$  of them. Bins containing items from  $B$  cannot accept any of these items, and only two can be put together since  $e \geq \frac{k}{3}$ . Thus,  $n - \frac{|B|}{2} + 2(n - l) \leq 2(n - |B|)$ . Solving for  $l$ , we get  $l \geq \frac{n}{2} + \frac{3}{4}|B|$ . This shows that at least  $\frac{n}{2} - \frac{|B|}{4}$  bins to the left of  $l$  contain two or more items. By Lemma 5,  $|B| \leq n - 3$ . Thus,  $\frac{n}{2} - \frac{|B|}{4} \geq \frac{n}{2} - \frac{n-3}{4} = \frac{n+3}{4} \geq 3$ , since  $n \geq 9$ .  $\square$

**Lemma 7.** Assume that  $n \geq 9$ ,  $s \leq \frac{k}{2}$ , and  $\frac{|A|}{|A|+|R|} < \frac{2}{3}$ . Then,  $E_{UFF} - E_{OPT} < (n - |X|)\frac{k}{3} + \frac{k}{2}$ .

*Proof.* In the case where  $e \leq \frac{k}{3}$ , we have an upper bound of  $\frac{k}{3}$  on the average empty space in bins with one item as well as bins with more items. Thus,  $E_{UFF} - E_{OPT} \leq (n - |X|)\frac{k}{3}$ . Now, assume that  $e > \frac{k}{3}$ . First we show an upper bound on  $l$ . At time  $t$  no two bins can contain only one small item each. Therefore,  $|A_t| \geq 2l - |B| - 1$ . The total number of large items is  $|R_b| + |B| \geq \frac{1}{2}|A_t| - 1 + |B| \geq l + \frac{|B|}{2} - \frac{3}{2}$ . Since OPT must pack all these items in separate bins, we have  $l + \frac{|B|}{2} - \frac{3}{2} \leq n$ . Define  $z \geq 0$  such that  $n - l = z + \frac{|B|}{2} - \frac{3}{2}$ . Since every bin after bin  $l$  has two items of size greater than  $e$ , we have the following upper bound on the empty space in these  $n - l$  bins and the bins with an item from  $B \setminus X$ :  $e(|B| - |X|) + (k - 2e)(n - l) = e|B| - e|X| + (k - 2e)(z + \frac{|B|}{2} - \frac{3}{2}) < e|B| - \frac{k}{3}|X| + (k - 2e)\frac{|B|}{2} + (k - 2e)(z - \frac{3}{2}) = \frac{k|B|}{2} - \frac{k}{3}|X| + (k - 2e)(z - \frac{3}{2}) \leq \frac{k|B|}{2} - \frac{k}{3}|X| + (k - 2e)z < \frac{k|B|}{2} - \frac{k}{3}|X| + \frac{k}{3}z$ . Among the remaining bins,  $l - |B| = n - z - \frac{3|B|}{2} + \frac{3}{2}$  bins do not contain an item from  $X$ . All of these bins have at least two items, and according to Lemma 6 enough of these bins exist for us to conclude, by Lemma 4, that the empty space is at most  $\frac{k}{3}(n - z - \frac{3|B|}{2} + \frac{3}{2})$ . The total empty space is then less than  $\frac{k|B|}{2} - \frac{k}{3}|X| + \frac{k}{3}z + \frac{k}{3}(n - z - \frac{3|B|}{2} + \frac{3}{2}) = (n - |X| + \frac{3}{2})\frac{k}{3}$ .  $\square$

Then, by Lemma 7, if  $n \geq 9$ ,

$$\begin{aligned} |R_s| &\leq \frac{1}{s} \cdot \left( E_{UFF} - E_{OPT} - \frac{k}{2}|R_b| \right) < \frac{1}{s} \left( \frac{k}{3}(n - |X|) + \frac{k}{2} - \frac{k}{2}|R_b| \right) \\ &\leq n - |X| + \frac{3}{2} - \frac{3}{2}|R_b|. \end{aligned}$$

Using Lemma 3 as in Subcase 2a, we get

$$|R| < n - |X| + \frac{3}{2} - \frac{1}{2}(|L| + \frac{1}{2}|X| - 1) = n - \frac{5}{4}|X| - \frac{1}{2}|L| + 2, \text{ for } n \geq 9.$$

Thus,

$$\begin{aligned}
\frac{|A|}{|A| + |R|} &\geq \frac{2n - |L| - |X|}{2n - |L| - \frac{5}{4}|X| - \frac{1}{2} + 2} \\
&\geq \frac{2n - (|L| + |X|) + \frac{4}{3}}{3n - \frac{3}{2}(|L| + |X|) + 2} - \frac{\frac{4}{3}}{3n - \frac{3}{2}(|L| + |X|) + 2} \\
&= \frac{2}{3} - \frac{4}{6n + 3}, \text{ for } n \geq 9.
\end{aligned}$$

This bound is lower than the lower bounds obtained in Case 1 and Subcase 2a for all  $n$ . It is shown in [5] that  $\frac{7}{11}$  is an upper bound on FF's competitive ratio on accommodating sequences. For  $n \geq 22$ ,  $\frac{2}{3} - \frac{4}{6n+3} > \frac{7}{11}$ . Thus, for  $n \geq 22$ , UFF has a better competitive ratio than FF on accommodating sequences.  $\square$

*Remark:* It is easy to see that UFF's competitive ratio is  $\frac{1}{k}$ . If it is less than  $\frac{2}{3}$ , then  $R$  is nonempty, so at least  $n$  items are accepted. OPT can accept at most  $nk$  items, so the competitive ratio is at least  $\frac{1}{k}$ . For the upper bound, if  $\frac{3}{2}n$  items of size  $k$  followed by  $nk$  items of size 1 are given, UFF will accept  $n$  items of size  $k$ , while OPT will accept all of the small ones, giving a ratio of  $\frac{1}{k}$ . Note that this means that  $\mathcal{A}_{UFF}(\alpha) = \frac{1}{k}$ , for  $\alpha \geq \frac{5}{2}$ . Furthermore, if  $2n$  items of size  $\frac{k}{2}$  are given, followed by  $(\alpha - 1)nk$  items of size 1, UFF will accept  $2n$  items of size  $\frac{k}{2}$ , while OPT can accept  $2n + (\alpha - 1)n(k - 2)$  items, giving a ratio of  $\frac{2}{2 + (\alpha - 1)(k - 2)}$ . Thus, for any constant  $c > 0$ ,  $\mathcal{A}_{UFF}(\alpha) \in \Theta(\frac{1}{k})$ , if  $\alpha \geq 1 + c$ .

## 4 The Accommodating Function

Suppose that, for each sequence  $I$  of items, the on-line algorithm knows, beforehand, the number  $\alpha n$  of bins needed to pack the items in  $I$  (or a good upper bound on  $\alpha$ ). Then an accommodating function can be achieved for which the function value is constant (that is, independent of  $k$  and  $n$ ) when evaluated at a constant  $\alpha$ .

### 4.1 A Randomized Algorithm

One way of exploiting the extra knowledge is to use  $\alpha n$  “virtual” bins. At the beginning the randomized algorithm  $\mathbb{R}$  randomly decides which  $n$  of the  $\alpha n$  virtual bins are going to correspond to the “real”  $n$  bins. Call the set of these  $n$  virtual bins  $B_A$  and the rest of the  $\alpha n$  virtual bins  $B_R$ . An algorithm  $\mathbb{A}$  with a “good” competitive ratio on accommodating sequences  $AR_{\mathbb{A}}$  is used to decide where the actual items would be packed in the  $\alpha n$  virtual bins. When  $\mathbb{A}$  packs an item in a bin in  $B_A$ , the algorithm  $\mathbb{R}$  accepts the item and places it in the corresponding real bin. All other items are rejected.

The expected fraction of the items which  $\mathbb{R}$  accepts is at least  $\frac{AR_{\mathbb{A}}}{\alpha}$ , since on average  $\frac{|B_A|}{|B_A| + |B_R|} = \frac{n}{\alpha n} = \frac{1}{\alpha}$  of the items accepted by  $\mathbb{A}$  will be packed in  $B_A$ .

Using Unfair-First-Fit, this gives  $\mathcal{A}(\alpha) \geq \frac{2}{3\alpha}$  (asymptotically), which is constant when  $\alpha$  is.

Another way of using virtual bins is to use an algorithm that is known to be able to pack any 1-sequence of items in  $\beta n$  bins for some constant  $\beta$ . In this case,  $\alpha\beta n$  virtual bins are used. According to [7], for the algorithm Harmonic+1,  $\beta \leq 1.588720$ . Using Harmonic+1 for packing items in the virtual bins and randomly choosing the  $n$  bins for  $B_A$  gives  $\mathcal{A}(\alpha) \geq \frac{1}{1.58872\alpha} \approx \frac{0.629}{\alpha}$ . According to [7], even for randomized algorithms,  $\beta \geq 1.536$ . Since  $\frac{1}{1.536} \approx 0.651$ , this approach cannot give an accommodating function as good as the method described above using  $\alpha n$  virtual bins can.

*Remark:* Amos Fiat [11] has noted that the technique described above can be used more generally, for many maximization problems, to give good values for the accommodating function when  $\alpha$  is small. If an algorithm  $\mathbb{A}$  with competitive ratio on accommodating sequences  $AR_{\mathbb{A}}$  is used with a quantity  $\alpha n$  of the virtual resource, and a quantity  $n$  of these virtual resources are randomly chosen and used on the real resources, then the algorithm will achieve an accommodating function of  $\mathcal{A}(\alpha) \geq \frac{AR_{\mathbb{A}}}{\alpha}$ .

## 4.2 A Deterministic Algorithm

It is also possible for a deterministic algorithm to have an accommodating function such that the function value of the accommodating function is constant (that is, independent of  $k$  and  $n$ ) when evaluated at a constant  $\alpha$  as long as  $n \geq 5$ . The following algorithm  $\mathbb{D}$  has this property.

$\mathbb{D}$  divides the possible item sizes into  $\lceil \log_2 k \rceil$  intervals,  $S_1, S_2, \dots, S_{\lceil \log_2 k \rceil}$ , defined by  $S_1 = \{x \mid \frac{k}{2} \leq x \leq k\}$ , and  $S_i = \{x \mid \frac{k}{2^i} \leq x < \frac{k}{2^{i-1}}\}$ , for  $2 \leq i \leq \lceil \log_2 k \rceil$ . Thus, for any two items with sizes  $s_a$  and  $s_b$  belonging to the same size interval,  $s_a \geq \frac{1}{2}s_b$ .

For each  $i$ ,  $1 \leq i \leq \lceil \log_2 k \rceil$ ,  $\mathbb{D}$  does the following. It accepts the first item with size  $s \in S_i$ . After that it accepts every  $\frac{\alpha}{\beta}$ th item with size  $s \in S_i$ , for a given constant  $\beta$ , and rejects all other items with sizes in  $S_i$ . The accepted items are packed according to the First-Fit packing rule and the constant  $\beta$  will be chosen as described below, so that  $\mathbb{D}$  has no problem doing so. Since  $\mathbb{D}$  accepts every  $\frac{\alpha}{\beta}$ th item in each size interval,  $\mathcal{A}(\alpha) \geq \frac{\beta}{\alpha}$ .

Let  $O$  be the set of all the items given, let  $O_F$  be the set of items consisting of the first item in each size interval and let  $O' = O \setminus O_F$ . Let  $A$  be the set of items accepted by  $\mathbb{D}$  and let  $A' = A \setminus O_F$ . For any set  $S$  of items, let the volume of  $S$ , denoted by  $V(S)$ , be the sum of the sizes of the items in  $S$ .

It follows from Lemma 4 that the volume of the items in any First-Fit packing using  $n$  bins is more than  $\frac{nk}{2}$ . Thus, if  $\beta$  is chosen such that  $V(A) \leq \frac{nk}{2}$ ,  $\mathbb{D}$  will be able to pack all the accepted items.

To determine an appropriate value for  $\beta$ , first notice that  $V(O') \leq V(O) \leq \alpha nk$ , since all the items can fit in  $\alpha n$  bins, and  $V(O') > \frac{1}{2} \frac{\alpha}{\beta} V(A')$ , since for every item  $o \in A'$ ,  $\frac{\alpha}{\beta} - 1$  items, each of size  $s \geq \frac{1}{2}\text{size}(o)$ , have been rejected.

Combining these inequalities gives  $\frac{1}{2} \frac{\alpha}{\beta} V(A') < \alpha nk$ , and solving for  $V(A')$  yields  $V(A') < 2\beta nk$ .

$$\text{Furthermore, } V(O_F) \leq \sum_{i=0}^{\lceil \log_2 k \rceil - 1} \frac{k}{2^i} < \sum_{i=0}^{\infty} \frac{k}{2^i} = 2k.$$

We now have that  $V(A) = V(A') + V(O_F) < 2\beta nk + 2k$ . To obtain  $2\beta nk + 2k \leq \frac{nk}{2}$ ,  $n$  must be at least 5, for any  $\beta > 0$ . For  $n \geq 5$ ,  $\beta = \frac{1}{20}$  assures that  $V(A) \leq \frac{nk}{2}$ . If we accept that  $n$  must be at least 10, then  $\beta = \frac{3}{20}$  can be used. Thus, if  $n \geq 5$ ,  $\mathcal{A}(\alpha) \geq \frac{1}{20\alpha}$ , and if  $n \geq 10$ ,  $\mathcal{A}(\alpha) \geq \frac{3}{20\alpha}$ .

## References

1. B. Awerbuch, Y. Azar, and S. Plotkin. Throughput-Competitive On-Line Routing. In *34th IEEE Symposium on Foundations of Computer Science*, pages 32–40, 1993.
2. B. Awerbuch, Y. Bartal, A. Fiat, and A. Rosén. Competitive Non-Preemptive Call Control. In *Proc. 5th Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 312–320, 1994.
3. B. Awerbuch, R. Gawlick, T. Leighton, and Y. Rabani. On-line Admission Control and Circuit Routing for High Performance Computation and Communication. In *35th IEEE Symposium on Foundations of Computer Science*, pages 412–423, 1994.
4. J. Boyar and K. S. Larsen. The Seat Reservation Problem. *Algorithmica*, 25:403–417, 1999.
5. J. Boyar, K. S. Larsen, and M. N. Nielsen. The Accommodating Function — A Generalization of the Competitive Ratio. Tech. report 24, Department of Mathematics and Computer Science, University of Southern Denmark, Main Campus: Odense University, 1998. Extended version submitted for journal publication 1999.
6. J. Boyar, K. S. Larsen, and M. N. Nielsen. The Accommodating Function — A Generalization of the Competitive Ratio. In *Sixth International Workshop on Algorithms and Data Structures*, volume 1663 of *Lecture Notes in Computer Science*, pages 74–79. Springer-Verlag, 1999.
7. E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation Algorithms for Bin Packing: A Survey. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 2, pages 46–93. PWS Publishing Company, 1997.
8. E. G. Coffman, Jr., J. Y-T. Leung, and D. W. Ting. Bin Packing: Maximizing the Number of Pieces Packed. *Acta Informatica*, 9:263–271, 1978.
9. E. G. Coffman, Jr. and Joseph Y-T. Leung. Combinatorial Analysis of an Efficient Algorithm for Processor and Storage Allocation. *SIAM J. Comput.*, 8:202–217, 1979.
10. J. Csirik and G. Woeginger. On-Line Packing and Covering Problems. In Gerhard J. Woeginger Amos Fiat, editor, *Online Algorithms*, volume 1442 of *Lecture Notes in Computer Science*, chapter 7, pages 147–177. Springer-Verlag, 1998.
11. A. Fiat. Personal communication, 1999.

# A $d/2$ Approximation for Maximum Weight Independent Set in $d$ -Claw Free Graphs

Piotr Berman\*

Dept. of Computer Science and Engineering,  
The Pennsylvania State University

**Abstract.** In this paper we consider the following problem. Given is a  $d$ -claw free graph  $G = (V, E, w)$  where  $w : V \rightarrow \mathbf{R}_+$ . Our algorithm finds an independent set  $A$  such that  $w(A^*)/w(A) \leq d/2$  where  $A^*$  is an independent set that maximizes  $w(A^*)$ . The previous best polynomial time approximation algorithm obtained  $w(A^*)/w(A) \leq 2d/3$ .

## 1 Introduction

In an undirected graph a  $d$ -*claw*  $C$  is an induced subgraph that consists of an independent set  $T_C$  of  $d$  nodes, called *talons*, and the *center* node that is connected to all the talons. A graph is  $d$ -claw free if it possesses no  $d$ -claws. For convenience, we define 1-claw to be a singleton set  $C$  with  $T_C = C$ . We also define the center set of a claw  $C$  as  $Z_C = C - T_C$ .

The  $d$ -claw free graphs are studied for two reasons. One is that these graphs appear in many applications. In particular, we often consider graphs in which nodes the set of nodes is a family of sets, and edges indicate non-empty set intersections. If sets in the family have less than  $d$  elements, the graph are  $d$ -claw free. Other examples include families of oriented squares of unit size, which form 5-claw free graphs, and families of unit size circles which form 7-claw free graphs.

Another reason is that  $d$ -claw free graphs form the broadest natural family of graphs where algorithms for the Maximum Independent Set problem (MIS for short) have constant approximation ratio. Even this very simple algorithm assures ratio  $d - 1$  (we always assume that  $(V, E)$  is the input graph):

### definition

$N(K, L) = \{u \in L : \exists v \in K \text{ such that } \{u, v\} \in E \text{ or } u = v\}$

GREEDY

$A \leftarrow \emptyset$

**while**  $V - N(A, V) \neq \emptyset$

    choose  $u \in V - N(A, V)$

$A \leftarrow A \cup \{u\}$

---

\* Research supported by NSF grant CCR-9700053, [berman@cse.psu.edu](mailto:berman@cse.psu.edu)

One can generalize MIS problem by introducing a weight function  $w : V \rightarrow \mathbf{R}_+$ . The objective of  $w$ -MIS is to find an independent set  $A$  with maximum  $w(A)$ . The above GREEDY algorithm achieves the same approximation ratio for  $w$ -MIS, once we change the greedy selection as follows:

choose  $u \in V - N(A, V)$  with the maximum  $w(u)$

An obvious challenge is to find polynomial time algorithms with better approximation. A natural idea is to apply *small improvements*. We say that a node set  $C$  improves  $w(A)$  if  $w(A - N(C, A) \cup C) > w(A)$ . The following algorithm approximates MIS for d-claw free graph with ratio  $d/2$  (see [BNR]):

```
SIZETWOIMP
A ← ∅
while there exists {u, v} that improves |A|
    A ← A - N({u, v}, A) ∪ {u, v}
```

By increasing the size of allowed improvements, one can obtain polynomial time algorithms with ratios approaching  $(d-1)/2$  [HS]. However, it was not obvious how to extend this idea to  $w$ -MIS in d-claw free graphs. Recently, Chandra and Halldórsson [CH] have found that the following algorithm has ratio  $2/3$ <sup>1</sup>:

```
BESTIMP
A ← ∅
while there exists claw C such that  $T_C$  improves  $w(A)$ 
    if  $V - N(A, V) \neq \emptyset$ 
        choose  $u \in V - N(A, V)$  with maximum  $w(u)$ ,  $C \leftarrow \{u\}$ 
    else
        choose claw C that maximizes  $w(T_C)/w(N(T_C, A))$ 
    A ← A - N( $T_C$ , A) ∪  $T_C$ 
```

Chandra and Halldórsson show how to modify BESTIMP so it runs in polynomial time: (i) find an approximate solution  $A$  using GREEDY; (ii) rescale the weight function so that  $w(A) = k|V|$ ; (iii) run the algorithm BESTIMP for the weight function  $[w]$ . Because each iteration increases  $[w](A)$  by at least 1, and we cannot get  $[w](A) > w(A^*)$ , there are fewer than  $(d-1)k|V|$  iterations. In turn, in each iteration we inspect only a polynomial number of candidates for the claw  $C$ . This assures that the new algorithm runs in polynomial time. Moreover, the solution of the new algorithm satisfies  $w(A) \geq [w](A) \geq [w](A^*)/(2/3 d) > w(A^*)/(k/k-1 \times 2/3 d)$ , thus the approximation ratio increases by factor  $k/k-1$ .

In this paper, we analyze the following algorithm and show that it provides the same approximation ratio for  $w$ -MIS as SIZETWOIMP for MIS:

```
SQUAREIMP
A ← ∅
while there exists claw C such that  $T_C$  improves  $w^2(A)$ 
    A ← A - N( $T_C$ , A) ∪  $T_C$ 
```

---

<sup>1</sup> The analysis of Chandra and Halldórsson holds only for the graphs formed from sets with fewer than  $d$  elements.

## 2 Analysis of SQUAREIMP

One can extend the analysis of the running time of BESTIMP to SQUAREIMP. While it does not have a polynomial bound on the running time, we can modify this algorithm so that it runs in time  $O(k^2 p(|V|))$  for some fixed polynomial  $p$ , while increasing the approximation ratio by  $k/k-1$  factor. The only difference is that we have a different estimate on the number of iterations. In particular, after we rescale  $w$  in step (ii), we have  $w(A) = k|V|$  and consequently  $w^2(A) \leq k^2|V|^2$ ; because GREEDY for  $w$  is also GREEDY for  $w^2$ , this implies that  $w^2(A^*) \leq (d-1)k^2|V|^2$ ; thus our estimate on the number of iterations of the modified SQUAREIMP is higher than the estimate for the modified BESTIMP by  $k|V|$  factor.

To see that the approximation ratio of SQUAREIMP is at least  $d/2$ , we may construct a small example for each  $d$ , in which  $w(u) = 1$  for every node. The set of nodes in this example is a union of two independent sets,  $A$  and  $B$ . Set  $A$ , has  $d-1$  elements. Set  $B$  consists of subsets of  $A$  with 1 or 2 elements; thus  $|B| = (d-1)(d-2)/2 + d-1 = (d-1)d/2 = d/2|A|$ . For  $u \in A$  and  $v \in B$ ,  $\{u, v\}$  is an edge if and only if  $u \in v$ . Algorithm SQUAREIMP may start by picking, one by one, elements of set  $A$ . It is easy to see that subsequently SQUAREIMP terminates because no claw improves  $|A|$ .

The above example shows also that we cannot improve SQUAREIMP by replacing  $w^2$  with some other  $w^c$ .

To show that the approximation ratio is at most  $d/2$ , we will start from the analysis of algorithm WISHFUL THINKING. The name of this algorithm comes from the fact that it is quite obvious that it delivers the desired approximation ratio, however this claim holds under the assumption that it terminates. Later, instead of analyzing the running time of WISHFUL THINKING directly, we will show that it cannot make more iterations than SQUAREIMP while SQUAREIMP cannot have a larger approximation ratio.

### definition

$$N(u, A) = N(\{u\}, A)$$

$n(u)$  is a node  $v \in N(u, A)$  with the maximum value of  $w(v)$

$$\text{charge}(u, v) = \begin{cases} w(u) - 1/2 w(N(u, A)) & \text{if } v = n(u) \\ 0 & \text{otherwise} \end{cases}$$

$C$  is a good claw if either  $N(C, A) = \emptyset$  or

$$Z_C = \{v\} \subset A \text{ and } \sum_{u \in T_C} \text{charge}(u, v) > 1/2 w(v)$$

$C$  is a nice claw if it is a minimal set that is a good claw

### WISHFUL THINKING

$$A \leftarrow \emptyset$$

**while** there exists a nice claw  $C$

$$A \leftarrow A - N(T_C, A) \cup T_C$$

**Lemma 1.** *Assume that WISHFUL THINKING has terminated and that  $A^*$  is an independent set. Then  $w(A^*)/w(A) \leq d/2$ .*

*Proof.* We will distribute  $w(A^*)$  among the nodes of  $A$  in such a way that no node  $v \in A$  receives more than  $1/2 dw(v)$ . The distribution consists of two steps.

In the first,  $u \in A^*$  sends to each  $v \in N(u, A)$  a portion of its weight equal to  $\frac{1}{2}w(v)$ . Note that  $N(u, A)$  is non-empty, otherwise  $\{u\}$  is a nice claw. Also, in this step  $u$  sends a portion of its weight equal to  $\frac{1}{2}w(N(u, A))$ , consequently the portion of its weight that is not distributed yet equals  $\text{charge}(u, n(u))$ . In the second step  $u$  sends  $\text{charge}(u, n(u))$  to  $n(u)$ .

On the receiving side, in the first step a node  $v \in A$  gets  $\frac{1}{2}w(v)$  from every neighbor in  $A^*$ , and there are at most  $d - 1$  of them (otherwise they would form talons of a  $d$ -claw with center  $\{v\}$ ). Thus  $v$  gets at most  $(d/2 - 1/2)w(v)$  in the first step. Moreover,  $v$  gets at most  $\frac{1}{2}w(v)$  in the second step, otherwise the nodes that send positive charges to  $v$  form talons of a good claw, and such a claw cannot exists when WISHFUL THINKING terminates.

While the goal of WISHFUL THINKING algorithm is the maximization of  $w(A)$ , an iteration may actually decrease  $w(A)$ . Consider  $S = \{v_0, \dots, v_4\} \subset A$  and  $T = \{u_1, u_2\} \subset V - A$  and make the following assumptions:

- (a)  $n(u) = v_0$  for  $u \in T$ ,
- (b)  $w(u) = 18$  for  $u \in T$ ,
- (c)  $N(u_i, A) = \{v_0, v_{2i-1}, v_{2i}\}$  for  $i = 1, 2$ , and
- (d)  $w(v) = 10$  for  $v \in S$ .

One can see that  $\text{charge}(v, u_0) = 3$  for  $v \in T$  and that  $3 + 3 \geq \frac{1}{2}10$ , thus  $T \cup \{u_0\}$  is a nice claw. If we apply this claw to perform an iteration of WISHFUL THINKING,  $A$  changes into  $A - S \cup T$  and  $w(A)$  decreases by 12.

Because WISHFUL THINKING can alternate between increasing and decreasing  $w(A)$  we need the following lemma to show that it actually terminates.

**Lemma 2.** *If  $C$  is a nice claw, then  $T_C$  improves  $w^2(A)$ .*

*Proof.* We will use  $T$  to denote  $T_C$ . We need to show that  $w^2(A - N(T, A) \cup T) > w^2(A)$ .

Consider first the case when  $N(T, A) = \emptyset$ . In this case  $A - N(T, A) \cup T = A \cup T$  and the claim is obvious.

In the remaining case  $Z_C = \{v\} \subset A$ . We will develop a condition that implies that  $w^2(A - N(T, A) \cup T) > w^2(A)$ , and then we will show that if  $C$  is nice, then  $T$  satisfies this condition.

By subtracting  $w^2(A - N(T, A))$  from both sides of  $w^2(A - N(T, A) \cup T) > w^2(A)$  we get an equivalent inequality

$$w^2(T) > w^2(N(T, A)) \quad (1)$$

Observe that

$$N(T, A) = \{v\} \cup \bigcup_{u \in T} N(u, A - \{v\})$$

and therefore (1) is implied by

$$\begin{aligned} \sum_{u \in T} w^2(u) &> w^2(v) + \sum_{u \in T} w^2(N(u, A - \{v\})) &\equiv \\ \sum_{u \in T} w^2(u) - w^2(N(u, A - \{v\})) &> w^2(v) &\equiv \end{aligned}$$

$$\sum_{u \in T} \frac{w^2(u) - w^2(N(u, A - \{v\}))}{w(v)} > w(v) \quad (2)$$

Now we will show that (2) holds if  $T \cup \{v\}$  is a nice claw. Under this assumption,  $T$  is a minimal set such that

$$\sum_{u \in T} \text{charge}(u, v) > 1/2 w(v) \quad (3)$$

Because set  $T$  is minimal, every term on the left-hand side of (3) is positive, and in particular,  $n(u) = v$ . Thus to show (2) it suffices to show that

$$\frac{w^2(u) - w^2(N(u, A - \{v\}))}{w(v)} \geq 2 \times \text{charge}(u, v) \quad (4)$$

holds if  $\text{charge}(u, v) > 0$ . By the definition of  $\text{charge}$ , this is true if

$$\frac{w^2(u) - w^2(N(u, A - \{v\}))}{w(v)} \geq 2w(u) - w(N(u, A)) \quad (5)$$

holds whenever  $v$  is an element of  $N(u, A)$  with the maximum weight and  $2w(u) > w(N(u, A))$ .

If we replace the weight function  $w$  with  $cw$ , then both sides of (5) will be multiplied by  $c$  and this is an equivalent transformation. Therefore we may assume, for the ease of calculations, that  $w(N(u, A)) = 2$ . Because  $2w(u) > 2$ , we may assume that for some  $x > 0$  we have  $w(u) = 1 + x$ . In the proof of (5) we consider two cases.

**Case 1.**  $w(v) = 1 + y$  for some  $y \geq 0$ . Then  $w(N(u, A - \{v\})) = 1 - y$ , hence  $w^2(N(u, A - \{v\})) \leq (1 - y)^2$ . Therefore (5) is implied by

$$\frac{(1+x)^2 - (1-y)^2}{1+y} \geq 2x \equiv 1 + 2x + x^2 - 1 + 2y - y^2 \geq 2x + 2xy \equiv$$

$$x^2 + 2y \geq y^2 + 2xy \equiv x^2 - 2xy + y^2 \geq 2y^2 - 2y \equiv (x-y)^2 \geq -2y(1-y).$$

Because  $0 \leq y \leq 1$ , the last inequality is obvious.

**Case 2.**  $w(v) = 1 - y$  for some  $y \geq 0$ . Then  $w(N(u, A - \{v\})) = 1 + y$ , while the largest weight in  $N(u, A - \{v\})$  is at most  $1 - y$ , hence  $w^2(N(u, A - \{v\})) \leq (1 + y)(1 - y) = 1 - y^2$ . Therefore (5) is implied by

$$\frac{(1+x)^2 - (1-y)^2}{1-y} \geq 2x \equiv$$

$$1 + 2x + x^2 - 1 + y^2 \geq 2x - 2xy \equiv x^2 + y + 2 \geq -2xy.$$

Again, the last inequality is obvious.

Lemma 2 allows us to relate algorithms WISHLFUL THINKING and SQUAREIMP. Because each nice claw improves  $w^2(A)$ , a run of WISHLFUL THINKING forms the initial part of a run of SQUAREIMP. Consequently, the number of iteration

performed by WISHFUL THINKING is at most as large as the number of iterations of SQUAREIMP. In turn, when SQUAREIMP terminates, we obtain a candidate set  $A$  for which no claw improves  $w^2(A)$ , hence no nice claw may exist, hence this candidate independent set satisfies the assumption of Lemma 1.

If we compare the virtues of these two algorithms, SQUAREIMP has a more succinct formulation, while WISHFUL THINKING is more efficient: the searching space is smaller when we seek a nice claw than when we seek a claw that improves  $w^2(A)$ , therefore the time needed to perform an iteration is smaller in the case of WISHFUL THINKING.

To see that the searching space of WISHFUL THINKING is indeed smaller, note that we can approach it as follows. Given the current candidate  $A$ , for every node  $u \in V_A$  we can compute  $n(u)$  and  $\text{charge}(u, n(u))$ . Then for a given  $v \in A$  we need to inspect independent sets contained in  $n^{-1}(v)$  (possible sets of talons). Moreover, we exclude the nodes that do not have a positive charge, and when we evaluate a possible set of talons, we consider only its sum of charges, and we do not need to compute its set of neighbors in  $A$ .

Without going into details of the running time analysis we can formulate the following theorem:

**Theorem 1.** *For every  $d$  there exists an algorithm that given a  $d$ -claw free graph with  $n$  nodes and  $k > 1$ , finds a solution to  $w$ -MIS problem with approximation ratio  $\frac{k}{k-1} \frac{1}{2} d$  in time that is polynomial in  $kn$ .*

## References

- BNR. V. Bafna, B. Narayana and R. Ravi, *Non-overlapping local alignments (weighted independent sets of axis parallel rectangles*, WADS 1995, Springer-Verlag LNCS **955**:506-517, to appear in Disc. Appl. Math.
- CH. B. Chandra and M. M. Halldórsson, *Greedy local improvement and weighted packing approximation*, SODA 1999.
- HS. C. A. Hurkens and A. Schrijver, *One the size of systems of sets every  $t$  of which have an SDR, with an application to the worst-case ratio heuristics for packing problems*, SIAM J. Discr. Math. **2**(1):68-72, Feb. 1989.

# Approximation Algorithms for the Label-Cover<sub>MAX</sub> and Red-Blue Set Cover Problems

## (Extended abstract)

David Peleg \*

The Weizmann Institute of Science,  
Department of Computer Science and Applied Mathematics, Rehovot, 76100 Israel.  
[peleg@wisdom.weizmann.ac.il](mailto:peleg@wisdom.weizmann.ac.il)

**Abstract.** This paper presents approximation algorithms for two problems. First, a randomized algorithm guaranteeing approximation ratio  $\sqrt{n}$  with high probability is proposed for the Max-Rep problem of [Kor98], or the Label-Cover<sub>MAX</sub> problem (cf. [Hoc95]), where  $n$  is the number of vertices in the graph. This algorithm is then generalized into a  $4\sqrt{n}$ -ratio algorithm for the nonuniform version of the problem. Secondly, it is shown that the Red-Blue Set Cover problem of [CDKM00] can be approximated with ratio  $2\sqrt{n \log \beta}$ , where  $n$  is the number of sets and  $\beta$  is the number of blue elements. Both algorithms can be adapted to the weighted variants of the respective problems, yielding the same approximation ratios.

## 1 Introduction

### 1.1 Background

Recent classifications of NP-hard problems by their approximability properties have led to the identification of a group of problems termed *class III* problems in [Hoc95]. These problems can be informally characterized as ones known to have no approximation algorithm with ratio  $2^{\log^{1-\epsilon} n}$  (for any  $0 < \epsilon < 1$ ) under some plausible complexity-theoretic assumption (such as  $NP \neq P$  or  $NP \not\subseteq DTIME(n^{O(\text{polylog } n)})$ ). We henceforth refer to this property as *strong inapproximability*. This class includes problems such as the minimization and maximization versions of *Label-Cover* [ABSS93], *AND/OR Scheduling* [GM97], *Minimum-Monotone-Satisfying-Assignment (MMSA)* [ABMP98], *Min-Rep* and *Max-Rep* [Kor98], *Red-Blue Set Cover* [CDKM00], and more.

While negative (strong inapproximability) results are known for all of those problems (and indeed, in a certain sense they define the class), less is known about positive (approximability) results. The current paper is concerned with providing such results for some of the above problems.

---

\* Supported in part by a grant from the Israel Ministry of Science and Art.

## 1.2 The Problems Considered

**Max-Rep, Label-Cover<sub>MAX</sub> and related problems.** The *Max-Rep* problem is defined in [Kor98] as follows. We are given a bipartite graph  $G(U, W, E)$ , where  $U$  and  $W$  are each split into a disjoint union of  $k$  sets,  $U = \bigcup_{i=1}^k A_i$  and  $W = \bigcup_{i=1}^k B_i$ . The sets  $A_i, B_i$  all have size  $m$ . Let  $\mathcal{A} = \{A_1, \dots, A_k\}$  and  $\mathcal{B} = \{B_1, \dots, B_k\}$ . An instance of the problem consists of the 5-tuple  $(U, W, E, \mathcal{A}, \mathcal{B})$ . The bipartite graph  $G$  and the partitions  $\mathcal{A}$  and  $\mathcal{B}$  of  $U$  and  $W$  induce a bipartite super-graph  $\mathcal{H} = (\mathcal{A}, \mathcal{B}, E_{\mathcal{H}})$ . Two super-vertices  $A_i$  and  $B_j$  are adjacent in  $\mathcal{H}$  iff there exist some  $u \in A_i$  and  $w \in B_j$  which are adjacent in  $G$ .

A set of vertices  $C \subseteq U \cup W$  is said to *cover* the super-edge  $(A_i, B_j)$  if it contains a pair of vertices  $u, w$  such that  $u \in A_i, w \in B_j$  and  $(u, w) \in E$ . The set  $C$  is a *legal cover* for  $\mathcal{H}$  if it contains at most one vertex from each super-vertex. It is required to select a legal cover  $C$  for  $\mathcal{H}$  covering the maximum number of super-edges possible.

A minimization version of this problem, called *Min-Rep*, is also introduced in [Kor98]. In this version, a cover  $C$  must cover *every* super-edge, but it may contain any number of vertices from each super-vertex, and the goal is to select a minimum size cover  $C$  for  $\mathcal{H}$ .

A closely related problem is the *Label-Cover* problem, introduced in [ABSS93] and presented in [Hoc95] as one of six canonical problems for proving hardness of approximation. This problem has minimization and maximization versions called Label-Cover<sub>MIN</sub> and Label-Cover<sub>MAX</sub>, which can be represented as variants of the Min-Rep and Max-Rep problems respectively, except that the notion of super-edge coverage is slightly different. Namely, a super-edge  $(A_i, B_j)$  is said to be *covered* if for *every* vertex  $u \in A_i \cap C$  there is a vertex  $w \in B_j \cap C$  such that  $(u, w) \in E$ . Note that the Label-Cover<sub>MAX</sub> problem is equivalent to the Max-Rep problem.

**The Red-Blue Set Cover Problem.** The *Red-Blue Set Cover* problem was introduced in [CDKM00]. It is a natural generalization of the set-cover problem, defined as follows. Consider a finite universe partitioned into two disjoint sets,  $U = R \cup B$ , where  $R = \{r_1, \dots, r_\rho\}$  is a set of *red* elements and  $B = \{b_1, \dots, b_\beta\}$  is a set of *blue* elements. We are given a collection of sets over the universe  $U$ ,  $\mathcal{S} = \{S_1, S_2, \dots, S_n\}$ . For any subcollection  $\mathcal{S}' \subseteq \mathcal{S}$ , let

$$U(\mathcal{S}') = \bigcup_{S_i \in \mathcal{S}'} S_i, \quad B(\mathcal{S}') = U(\mathcal{S}') \cap B, \quad R(\mathcal{S}') = U(\mathcal{S}') \cap R.$$

The goal is to choose a subcollection  $\mathcal{S}'$  of  $\mathcal{S}$  that covers all the elements of  $B$  (i.e., s.t.  $B \subseteq B(\mathcal{S}')$ ) while minimizing  $|R(\mathcal{S}')|$ , the number of red elements in  $\mathcal{S}'$ .

The Red-Blue Set Cover problem is also shown in [CDKM00] to be equivalent to *MMSA*<sub>3</sub>, the third level of the *Minimum-Monotone-Satisfying-Assignment* problem introduced in [ABMP98].

### 1.3 Previous Work

The Label-Cover problem was introduced in [ABSS93], where it was also shown to be strongly inapproximable. More precisely, it was shown that it is quasi-NP-hard to approximate the problem with ratio  $2^{\log^{1-\epsilon} n}$  for any constant  $0 < \epsilon < 1$  (or in other words, such approximation is impossible unless  $NP \subseteq DTIME(n^{O(\text{polylog } n)})$ ). This result was recently improved in [DS99], by weakening the complexity-theoretic assumption to  $NP \neq P$  and allowing  $\epsilon$  to be as small as  $\log \log^{-c} n$  for any  $c < 1/2$ .

The Max-Rep and Min-Rep problems were introduced in [Kor98], where it was also shown that both are strongly inapproximable.

The strong inapproximability of the Red-Blue Set Cover problem was shown independently in [CDKM00] and [EP00]. Applications of the Red-Blue Set Cover problem in a variety of domains, such as data mining applications, information retrieval or general machine learning and classification, are discussed at length in [CDKM00], as well as a number of special-case variants of the problem and related problems, including Set Cover, Group Steiner and Directed Steiner, and Minimum Color Path.

Few positive results exist for the above problems. The Red-Blue Set Cover problem admits naive approximation algorithms with ratios  $\beta$ ,  $\rho$ , or  $n \log \beta$ . A number of better approximation algorithms are given in [CDKM00] for this problem. Specifically, letting  $k_B$  (respectively,  $k_R$ ) denote the maximum number of blue (resp., red) elements in any of the sets  $S_i$ , the paper presents approximation algorithms with ratio  $2\sqrt{k_B \cdot n}$  or  $O(n^{1-1/k_R} \log n)$ . Hence these algorithms are efficient when  $k_B$  or  $k_R$  are small, but their approximation ratio may be as high as  $\Omega(\sqrt{n\beta})$  or  $\Omega(n \log n)$ , respectively, in the general case.

In [EP00] it is shown that the Min-Rep and Label-Cover<sub>MIN</sub> problems admit a  $\sqrt{n}$ -approximation ratio. It is also shown that the Min-Rep and Label-Cover<sub>MIN</sub> problems restricted to the cases where the girth of the induced supergraph is greater than  $t$ , admit an  $n^{2/t}$  approximation ratio. In particular, it follows that the Min-Rep and the Label-Cover<sub>MIN</sub> problems with girth greater than  $\log^\epsilon n$  (for some constant  $\epsilon > 0$ ) are not strongly inapproximable.

### 1.4 Contributions

The current paper presents approximation algorithms for two of the above problems.

The first algorithm, presented in Section 2, is a randomized algorithm guaranteeing approximation ratio  $\sqrt{n}$  with high probability for the Max-Rep problem (or for Label-Cover<sub>MAX</sub>). (A simple *deterministic* variant was recently pointed out by Y. Hassin [Has00].) This algorithm is then generalized into a  $4\sqrt{n}$ -ratio algorithm for the nonuniform version of the problem, in which there may be a different number of sets in  $\mathcal{A}$  and  $\mathcal{B}$ , and these sets may have different sizes. The algorithm can be generalized also to the weighted version of the problem, where super-edges have real nonnegative weights, and the goal is to maximize the total weight of covered super-edges.

In Section 3 we present an algorithm with approximation ratio  $2\sqrt{n \log \beta}$  for the Red-Blue Set Cover problem. The algorithm can be generalized also to the weighted version of the problem, in which every red element  $r_i \in R$  has a positive real weight associated with it, and the goal is to minimize the weight of the selected cover.

## 2 An Approximation Algorithm for the Max-Rep Problem

### 2.1 The Uniform Case

Let us start with some terminology. For every  $1 \leq i \leq k$ , let  $A_i = \{u_i^1, \dots, u_i^m\}$  and  $B_i = \{w_i^1, \dots, w_i^m\}$ . We think of the graph as drawn with the vertices of  $U$  on the left and the vertices of  $W$  on the right.

Consider a cover  $C$ . Without loss of generality we may assume that  $C = U^C \cup W^C$ , where  $U^C$  (respectively,  $W^C$ ) contains exactly one vertex  $u_i^C$  (resp.,  $w_i^C$ ) in each super-vertex  $A_i$  (resp.,  $B_i$ ) on the left (resp., right). In particular, we denote by  $C^* = U^* \cup W^*$  the optimal solution to the problem, and let  $u_i^*$  (resp.,  $w_i^*$ ) denote the unique vertex of  $U^* \cap A_i$  (resp.,  $W^* \cap B_i$ ).

For vertex subsets  $U' \subseteq U$  and  $W' \subseteq W$ , let  $G(U', W')$  denote the subgraph of  $G$  induced by  $U'$  and  $W'$ . For a vertex  $u \in U'$  (resp.,  $w \in W'$ ), let  $\deg(u, W')$  (resp.,  $\deg(w, U')$ ) denote its degree in  $G(U', W')$ . For  $u \in A_l$ , let  $\Gamma(u, \mathcal{H})$  denote the set of super-vertices  $B_i$  neighboring  $u$ , namely, such that there is an edge  $(u, w_i^j) \in E$  for some  $w_i^j \in B_i$ . Let  $sdeg(u, \mathcal{H})$  denote the super-degree of  $u$ , namely, the cardinality of  $\Gamma(u, \mathcal{H})$ . The super-degree of a vertex represents the number of super-edges it can potentially cover.

For any cover  $C$ , let  $\hat{E}(C)$  denote the set of super-edges of  $\mathcal{H}$  covered by  $C$ . Note that these are precisely the super-edges corresponding to the edges of the graph  $G(U^C, W^C)$ . Denote the cardinality of this set by  $f(C) = |\hat{E}(C)|$ .

We first present two approximation procedures for the problem. The first of the two has approximation ratio  $k$ , so it applies well in case there are few sets.

#### Procedure FEW\_SETS.

1. Calculate the super-degree  $sdeg(u, \mathcal{H})$  of every vertex  $u \in U$ .
2. Find the vertex  $\hat{u} \in U$  with maximum super-degree.
3. Construct a set  $\hat{W}$  consisting of one neighbor  $w_i^{j_i}$  of  $\hat{u}$  in every super-vertex  $B_i \in \Gamma(\hat{u}, \mathcal{H})$ .
4. Complete the set  $\hat{W} \cup \{\hat{u}\}$  into a cover  $\hat{C}$  arbitrarily.
5. Output the cover  $\hat{C}$ .

**Lemma 1.** *Procedure FEW\_SETS yields a  $k$ -approximation for the Max-Rep problem.*

*Proof.* By the choice of  $\hat{u}$ ,  $\text{sdeg}(u_i^*, \mathcal{H}) \leq f(\{\hat{u}\} \cup \hat{W}) \leq f(\hat{C})$  for every  $1 \leq i \leq k$ . Subsequently,

$$f(C^*) \leq \sum_{1 \leq i \leq k} \text{sdeg}(u_i^*, \mathcal{H}) \leq k \cdot f(\hat{C}). \quad \blacksquare$$

Our second approximation procedure has approximation ratio  $2m$ , so it applies well in case the sets are small.

**Procedure** SMALL\_SETS.

1. For every  $1 \leq i \leq k$ , draw a vertex  $\tilde{u}_i \in A_i$  uniformly at random.
2. Let  $\tilde{U} = \{\tilde{u}_1, \dots, \tilde{u}_k\}$ .
3. For every  $1 \leq i \leq k$  do:
  - (a) Compute the degree  $\deg(w_i^j, \tilde{U})$  for every vertex  $w_i^j \in W_i$ .
  - (b) Let  $\tilde{w}_i$  be the vertex with maximum degree.
4. Let  $\tilde{W} = \{\tilde{w}_1, \dots, \tilde{w}_k\}$ .
5. Output the cover  $\tilde{C} = (\tilde{U}, \tilde{W})$ .

For the analysis, we need to argue that the cover  $\tilde{C} = (\tilde{U}, \tilde{W})$  constructed by Procedure SMALL\_SETS is not much worse than the optimal cover  $C^* = U^* \cup W^*$ . To do that, let us first consider the intermediary “mixed” cover  $\bar{C} = (\tilde{U}, W^*)$ .

By the choice of  $\tilde{W}$ , it is clear that once  $\tilde{U}$  is fixed,  $W^*$  is no better than  $\tilde{W}$ . Hence comparing  $\tilde{C}$  to  $\bar{C}$ , the following claim is immediate.

**Lemma 2.**  $f(\tilde{C}) \geq f(\bar{C})$ .  $\blacksquare$

On the other hand, comparing  $\bar{C}$  to  $C^*$  we have:

**Lemma 3.**  $\mathbb{E}(f(\bar{C})) \geq \frac{1}{m} \cdot f(C^*)$ .

*Proof.* Let  $\tilde{d}_i = \deg(\tilde{u}_i, W^*)$  and  $d_i^* = \deg(u_i^*, W^*)$ . Observe that if  $\tilde{u}_i = v_i^*$  then  $\tilde{d}_i = d_i^*$ . As this happens with probability  $1/m$ , we have that  $\mathbb{E}(\tilde{d}_i) \geq \frac{1}{m} \cdot d_i^*$ .

Noting that  $f(C^*) = \sum_{i=1}^k d_i^*$  and  $f(\bar{C}) = \sum_{i=1}^k \tilde{d}_i$ , we conclude that

$$\mathbb{E}(f(\bar{C})) = \sum_{i=1}^k \mathbb{E}(\tilde{d}_i) \geq \frac{1}{m} \sum_{i=1}^k d_i^* = \frac{1}{m} \cdot f(C^*). \quad \blacksquare$$

**Corollary 1.**  $\mathbb{E}(f(\tilde{C})) \geq \frac{1}{m} \cdot f(C^*)$ .  $\blacksquare$

To get this result with high probability, we apply the following procedure.

**Procedure** SMALL\_SETS\_2.

1. Set  $\ell = 2m \log n$ .
2. Invoke Procedure SMALL\_SETS for  $\ell$  times.
3. Select the best result.

We rely on the following elementary fact.

**Lemma 4.** *If  $X$  is a random variable in the range  $[0, m\alpha]$  with expectation  $\mathbb{E}(X) = \alpha$ , then the probability that  $X \leq \alpha/2$  is at most  $1 - \frac{1}{2m}$ .*

By applying the above fact to the random variable  $f(\tilde{C})$  with  $\alpha = f(C^*)/m$ , we get that in each invocation of Procedure SMALL\_SETS, the probability that the gain of the resulting cover  $\tilde{C}$  is  $f(\tilde{C}) \leq f(C^*)/2m$  is at most  $1 - \frac{1}{2m}$ . Subsequently, the probability that the gain of none of the  $\ell$  covers exceeds  $f(C^*)/2m$  is at most  $(1 - \frac{1}{2m})^\ell \approx 1/n$ .

**Corollary 2.** *With probability at least  $1 - 1/n$ , Procedure SMALL\_SETS\_2 yields a  $2m$ -approximation for the Max-Rep problem.* ■

Finally, by combining Lemma 1 and Corollary 2 we conclude that applying both procedures FEW\_SETS and SMALL\_SETS\_2 and selecting the better result yields an approximation ratio of  $\min\{k, 2m\}$ . As  $n = 2km$ , either  $k \leq \sqrt{n}$  or  $2m \leq \sqrt{n}$  must hold, hence we have the following.

**Theorem 1.** *There is a randomized algorithm yielding an approximation with ratio  $\sqrt{n}$  with probability at least  $1 - 1/n$  for the Max-Rep problem.* ■

Let us remark that a simple *deterministic* variant of Procedure SMALL\_SETS, and hence of the entire algorithm, was recently pointed out by Y. Hassin [Has00].

## 2.2 The Nonuniform Case

Let us now generalize the approximation algorithm to the case where the partitioning of the graph is nonuniform, i.e., there are  $k_U$  sets  $A_i$  and  $k_W$  sets  $B_i$ , and each of those sets is possibly of different size.

It is easy to verify that the procedures described earlier still work correctly, albeit with weaker approximation ratios. In particular, Procedure FEW\_SETS will guarantee approximation ratio at most  $k_U$ . Analogously, a dual procedure FEW\_SETS\_2 which reverses the roles of the sets  $U$  and  $W$  (i.e., selects the best vertex  $\hat{w} \in W$  and bases the cover on  $\hat{w}$  and its neighbors in  $U$ ) will yield ratio  $k_W$ . Procedure SMALL\_SETS\_2 will guarantee approximation ratio at most  $\hat{m}(\mathcal{H}) = \max\{|A_i| \mid 1 \leq i \leq k_U\}$ . Unfortunately, in a nonuniform instance, all of these bounds might be simultaneously as large as  $\Omega(n)$ .

To get a better bound, we partition the problem into four subproblems as follows. First, split the sets  $A_i$  and  $B_i$  into *large* and *small* ones, letting

$$\begin{aligned} \mathcal{A}_L &= \{A_i \mid |A_i| \geq \sqrt{n}\}, & \mathcal{A}_S &= \{A_i \mid |A_i| < \sqrt{n}\}, \\ \mathcal{B}_L &= \{B_i \mid |B_i| \geq \sqrt{n}\}, & \mathcal{B}_S &= \{B_i \mid |B_i| < \sqrt{n}\}, \end{aligned}$$

and taking

$$\begin{aligned} U_L &= \cup_{A_i \in \mathcal{A}_L} A_i, & U_S &= U \setminus U_L, \\ W_L &= \cup_{B_i \in \mathcal{B}_L} B_i, & W_S &= W \setminus W_L. \end{aligned}$$

The edge set  $E$  is partitioned accordingly into four subsets

$$E_{XY} = E \cap U_X \times W_Y, \quad \text{for } X, Y \in \{L, S\}.$$

The problem now splits into four subproblems, denoted  $\Pi_{XY}$  for  $X, Y \in \{L, S\}$ , where  $\Pi_{XY}$  is defined over the graph  $G_{XY} = (U_X, W_Y, E_{XY})$  and the super-graph  $\mathcal{H}_{XY}$  induced by the super-vertices of  $\mathcal{A}_X$  and  $\mathcal{B}_Y$ .

Clearly, each cover  $C$  for the original problem induces four covers  $C_{XY}$  for the subproblems, with

$$f(C) = f(C_{LL}) + f(C_{LS}) + f(C_{SL}) + f(C_{SS}).$$

Subsequently, if it is possible to approximate *each* of the four subproblems  $\Pi_{XY}$  for  $X, Y \in \{L, S\}$  *separately*, giving it a cover  $C_{XY}$  with ratio at most  $\gamma$ , then we can guarantee an approximation for the original problem with ratio at most  $4\gamma$ , simply by taking the largest of the resulting four covers and completing it arbitrarily.

The crucial observation is that on subproblems  $\Pi_{LL}$  and  $\Pi_{LS}$  we have  $k_{U_L} \leq \sqrt{n}$ , so Procedure FEW\_SETS yields approximation ratio  $\sqrt{n}$ . Likewise, on subproblem  $\Pi_{SL}$  we have  $k_{W_L} \leq \sqrt{n}$ , so the dual Procedure FEW\_SETS\_2 again yields approximation ratio  $\sqrt{n}$ . Finally, on subproblem  $\Pi_{SS}$  we have  $\hat{m}(\mathcal{H}_{SS}) \leq \sqrt{n}$ , so Procedure SMALL\_SETS\_2 will yield approximation ratio  $\sqrt{n}$ .

**Theorem 2.** *There is a randomized algorithm yielding an approximation with ratio  $4\sqrt{n}$  with probability at least  $1 - 1/n$  for the Nonuniform Max-Rep problem.*

■

### 2.3 The Weighted Problem

Finally, let us consider the weighted variant of the problem, in which every super-edge  $(A_i, B_j)$  has a nonnegative real weight  $\omega(A_i, B_j)$  associated with it, and the goal is to maximize the weight of the selected cover. In the full paper we show that Procedures FEW\_SETS, FEW\_SETS\_2, SMALL\_SETS and SMALL\_SETS\_2 can be extended to the weighted setting with no change in the approximation ratio. This is done by defining appropriate generalizations of the deg and sdeg functions which take the super-edge weights into account. In particular, for a set  $E'$  of super-edges, let  $\omega(E') = \sum_{e \in E'} \omega(e)$ . Let sdeg( $u, \mathcal{H}$ ) denote the *weighted* super-degree of  $u$  in  $\mathcal{H}$ , namely, the total weight  $\omega(\{(A_l, B_i) \mid B_i \in \Gamma(u, \mathcal{H})\})$ . The super-degree of a vertex now represents the total weight of the super-edges it can potentially cover. Denote the weight of the set  $\hat{E}(C)$  by  $f(C) = \omega(\hat{E}(C))$ . Finally, for every  $w \in W'$ , let  $B(w)$  denote the set to which  $w$  belongs. For  $u \in A_l$ ,

denote the set of neighbors of  $u$  in  $G$  by  $\Gamma(u, G)$ . Then define  $\deg(u, W') = \sum_{w \in W' \cap \Gamma(u, G)} \omega(A_i, B(w))$ .

With these definitions, the entire analysis goes through with little change. We thus get the following. ■

**Theorem 3.** *There is a randomized algorithm yielding an approximation with ratio  $4\sqrt{n}$  with probability at least  $1 - 1/n$  for the Nonuniform Weighted Max-Rep problem.* ■

### 3 An Approximation Algorithm for the Red-Blue Set Cover Problem

Let us first give some definitions. For every red element  $r_i \in R$  and set collection  $\mathcal{S}$ , let  $\deg(r_i, \mathcal{S})$  denote the number of sets in  $\mathcal{S}$  that contain  $r_i$ . Let  $\Delta(\mathcal{S}) = \max\{\deg(r_i, \mathcal{S}) \mid r_i \in R\}$ .

For a set  $S_i$ , a set collection  $\mathcal{S}$  and a subset  $R' \subseteq R$  of red elements, denote the set obtained by discarding the elements of  $R'$  from  $S_i$  by  $\Phi(S_i, R') = S_i \setminus R'$ , and let

$$\Phi(\mathcal{S}, R') = \{\Phi(S_i, R') \mid S_i \in \mathcal{S}'\}.$$

For every set  $S_i \in \mathcal{S}$ , let  $r(S_i) = |R(\{S_i\})|$ , and for every subcollection  $\mathcal{S}' \subseteq \mathcal{S}$ , let  $r(\mathcal{S}') = |R(\mathcal{S}')|$ .

Let  $\mathcal{S}^*$  denote the optimal solution for the Red-Blue Set Cover problem on the instance  $\mathcal{S}$ .

#### 3.1 The Greedy Procedure

We make use of the following approximation procedure for the Red-Blue Set Cover problem.

##### Procedure GREEDY\_RB.

1. Modify  $\mathcal{S}$  into an instance  $\mathcal{T}$  of the weighted set cover problem as follows.
  - (a) Take  $\mathcal{T} = \Phi(\mathcal{S}, R)$ ,
  - (b) Assign each set  $T_i = \Phi(S_i, R)$  in  $\mathcal{T}$  a weight  $\omega(T_i) = r(S_i)$ .
2. Apply the greedy algorithm for weighted set cover to  $\mathcal{T}$ , and generate a cover  $\tilde{\mathcal{T}}$ .
3. Take the corresponding collection  $\tilde{\mathcal{S}} = \{S_i \mid T_i \in \tilde{\mathcal{T}}\}$  as the resulting approximation.

For every subcollection  $\mathcal{T}'$  of an instance  $\mathcal{T}$  of the weighted set cover problem, let  $\omega(\mathcal{T}') = \sum_{T_i \in \mathcal{T}'} \omega(T_i)$ . It is easy to verify the following.

**Lemma 5.** *For any set collection  $\mathcal{S}'$  and corresponding instance  $\mathcal{T}' = \Phi(\mathcal{S}', R)$  of the weighted set cover problem,*

$$r(\mathcal{S}') \leq \omega(\mathcal{T}') \leq \Delta(\mathcal{S}) \cdot r(\mathcal{S}').$$

*Proof.* Note that

$$\omega(\mathcal{T}') = \sum_{T_i \in \mathcal{T}'} \omega(T_i) = \sum_{S_i \in \mathcal{S}'} r(S_i) = \sum_{r_j \in R(\mathcal{S}')} \deg(r_j, \mathcal{S}') .$$

As  $1 \leq \deg(r_j, \mathcal{S}') \leq \Delta(\mathcal{S}') \leq \Delta(\mathcal{S})$  for every  $r_j \in R(\mathcal{S}')$ , we get

$$r(\mathcal{S}') \leq \sum_{r_j \in R(\mathcal{S}')} \deg(r_j, \mathcal{S}') \leq \Delta(\mathcal{S}) \cdot r(\mathcal{S}') ,$$

implying the claim.  $\blacksquare$

**Lemma 6.** *Procedure GREEDY\_RB has an approximation ratio of  $\Delta(\mathcal{S}) \cdot \log \beta$ .*

*Proof.* Denote the minimum-weight set cover for  $\mathcal{T}$  by  $\mathcal{T}^\#$ , and let  $\mathcal{T}^* = \Phi(\mathcal{S}^*, R)$  be the instance of the weighted set cover problem corresponding to  $\mathcal{S}^*$ . (Note that  $\mathcal{T}^\#$  and  $\mathcal{T}^*$  need not necessarily be the same.) It is known that the greedy algorithm yields a  $\log \beta$  approximation for the weighted set cover problem, namely,  $\omega(\tilde{\mathcal{T}}) \leq \log \beta \cdot \omega(\mathcal{T}^\#)$  [Chv79]. Therefore, by Lemma 5,

$$r(\tilde{\mathcal{S}}) \leq \omega(\tilde{\mathcal{T}}) \leq \log \beta \cdot \omega(\mathcal{T}^\#) .$$

The optimality of  $\mathcal{T}^\#$  implies that  $\omega(\mathcal{T}^\#) \leq \omega(\mathcal{T}^*)$ . Combined, we get that

$$r(\tilde{\mathcal{S}}) \leq \log \beta \cdot \omega(\mathcal{T}^*) .$$

Applying Lemma 5 again we get that

$$r(\tilde{\mathcal{S}}) \leq \log \beta \cdot \Delta(\mathcal{S}) \cdot r(\mathcal{S}^*) .$$

$\blacksquare$

### 3.2 The Main Procedure

For an integer parameter  $X$ , we consider the following procedure.

**Procedure LOW\_DEG( $X$ ).**

1. Discard from  $\mathcal{S}$  the sets with more than  $X$  red elements, setting  $\mathcal{S}_X \leftarrow \{S_i \in \mathcal{S} \mid r(S_i) \leq X\}$ .
2. If  $B(\mathcal{S}_X) \neq B$  then return  $\mathcal{S}$ . /\*  $\mathcal{S}_X$  is not feasible \*/
3. Set  $Y = \sqrt{n/\log \beta}$
4. Separate the red elements into *high* and *low* degree elements, setting  $R_H \leftarrow \{r_i \in R \mid \deg(r_i, \mathcal{S}_X) > Y\}$  and  $R_L \leftarrow R \setminus R_H$ .
5. Discard the elements of  $R_H$  from  $\mathcal{S}_X$ , setting  $\mathcal{S}_{X,Y} \leftarrow \Phi(\mathcal{S}_X, R_H)$ .
6. Apply Procedure GREEDY\_RB to  $\mathcal{S}_{X,Y}$ , and obtain a solution  $\tilde{\mathcal{S}}_{X,Y}$ .
7. Complete the sets of  $\tilde{\mathcal{S}}_{X,Y}$  into the corresponding sets of  $\mathcal{S}_X$  (by adding to each set  $T_i \in \tilde{\mathcal{S}}_{X,Y}$  originally obtained from  $S_i \in \mathcal{S}_X$  the discarded elements  $S_i \cap R_H$ ),  
and return the resulting solution  $\tilde{\mathcal{S}}_X$ .

**Lemma 7.**  $|R_H| \leq \sqrt{n \log \beta} \cdot X$ .

*Proof.* Each set  $S_i \in \mathcal{S}_X$  has at most  $X$  red elements. Hence

$$|R_H| \cdot Y < \sum_{r_j \in R_H} \deg(r_j, \mathcal{S}_X) \leq \sum_{r_j \in R} \deg(r_j, \mathcal{S}_X) = \sum_{S_i \in \mathcal{S}_X} r(S_i) \leq |\mathcal{S}_X| \cdot X \leq nX,$$

so  $|R_H| \leq nX/Y$ , implying the lemma. ■

### 3.3 The Approximation Algorithm

Now let us set  $\hat{X} = \max\{r(S_i^*) \mid S_i^* \in \mathcal{S}^*\}$ , and consider the performance of Procedure LOW\_DEG when invoked with the parameter  $X = \hat{X}$ .

**Lemma 8.** *Procedure LOW\_DEG( $\hat{X}$ ) yields an approximation ratio of at most  $2\sqrt{n \log \beta}$ .*

*Proof.* First observe, that  $\mathcal{S}_{\hat{X}}$  is necessarily feasible. Hence the procedure will always return a solution in its Step 7 (and not Step 2).

Let  $\mathcal{S}^*$  be some optimal solution for the problem, and let  $r_H^* = |R(\mathcal{S}^*) \cap R_H|$  and  $r_L^* = |R(\mathcal{S}^*) \cap R_L|$ . Since  $\Delta(\mathcal{S}_{\hat{X}}, Y) \leq Y$ , Lemma 6 guarantees that the solution produced by Procedure LOW\_DEG( $\hat{X}$ ) uses at most  $Y \cdot \log \beta \cdot r_L^* = \sqrt{n \log \beta} \cdot r_L^*$  red elements of  $R_L^*$ . By Lemma 7, the number of red elements of  $R_H^*$  contained in the solution generated by the procedure is at most  $\sqrt{n \log \beta} \cdot \hat{X}$ . Combined, the total number of red elements used by the procedure satisfies  $r(\tilde{\mathcal{S}}_{\hat{X}}) \leq \sqrt{n \log \beta} \cdot r_L^* + \sqrt{n \log \beta} \cdot \hat{X}$ . But by the definition of  $\hat{X}$ , necessarily  $r(\mathcal{S}^*) \geq \hat{X}$ , and hence  $r(\tilde{\mathcal{S}}_{\hat{X}}) \leq 2\sqrt{n \log \beta} \cdot r(\mathcal{S}^*)$ , yielding the lemma. ■

As  $\hat{X}$  is not known to us in advance, it will be necessary to search for it. This yields our final algorithm.

#### Algorithm LOW\_DEG2.

1. **For**  $X = 1$  to  $\rho$  **do:**  
    Invoke Procedure LOW\_DEG( $X$ ).  
2. Take the best of the obtained solutions.

**Theorem 4.** *Algorithm LOW\_DEG2 yields an approximation ratio of  $2\sqrt{n \log \beta}$  for the Red-Blue Set Cover problem.* ■

A minor variant of this algorithm yields an approximation ratio of  $(n\rho)^{1/3}$ . However, the problem clearly admits also a trivial approximation algorithm of ratio  $\rho$ , and  $(n\rho)^{1/3}$  is always dominated by the smaller of  $\rho$  and  $\sqrt{n}$ , so this variant is not as interesting (assuming the factor of  $\log \beta$  is negligible compared to the other terms).

### 3.4 The Weighted Case

Finally, let us consider the weighted variant of the problem, in which every red element  $r_i \in R$  has a positive real weight  $\omega(r_i)$  associated with it, and the goal is to minimize the weight of the selected cover. In the full paper we show that Procedures GREEDY\_RB and LOW\_DEG can be extended to the weighted setting with no change in the approximation ratio. In particular, in addition to the previous definitions, define the weight of a set  $S_i$  to be  $\omega(S_i) = \sum_{r_j \in S_i} \omega(r_j)$ , and for a subcollection  $\mathcal{S}'$  let  $\omega(\mathcal{S}') = \sum_{r_j \in R(\mathcal{S}')} \omega(r_j)$ . In Procedure GREEDY\_RB, Step 1(b) should assign each set  $T_i$  in  $\mathcal{T}$  the weight  $\omega(T_i) = \omega(S_i)$ . The inequalities of Lemma 5 become

$$\omega(\mathcal{S}') \leq \omega(\mathcal{T}') \leq \Delta(\mathcal{S}) \cdot \omega(\mathcal{S}') ,$$

with minimal changes in the proof, as well as in the proof of Lemma 6. In Procedure LOW\_DEG, the definition of  $\mathcal{S}_X$  changes to  $\mathcal{S}_X \leftarrow \{S_i \in \mathcal{S} \mid \omega(S_i) \leq X\}$ . As a result, Lemma 7 now asserts that  $\omega(R_H) \leq \sqrt{n \log \beta} \cdot X$ . The definition of  $\hat{X}$  becomes  $\hat{X} = \max\{\omega(S_i^*) \mid S_i^* \in \mathcal{S}^*\}$ . The proof of Lemma 8 uses  $\omega_H^* = \omega(R(\mathcal{S}^*) \cap R_H)$  and  $\omega_L^* = \omega(R(\mathcal{S}^*) \cap R_L)$  instead of  $r_H^*$  and  $r_L^*$ , respectively. We thus get the following.

**Theorem 5.** *There is an algorithm with approximation ratio  $2\sqrt{n \log \beta}$  for the Weighted Red-Blue Set Cover problem.* ■

### Acknowledgements

I am grateful to Michael Elkin, Yehuda Hassin and Hadas Taubman for helpful discussions.

### References

- ABMP98. M. Alekhnovich, S. Buss, S. Moran, and T. Pitassi. Minimum propositional proof length is NP-hard to linearly approximate. Manuscript, 1998.
- ABSS93. S. Arora, L. Babai, J. Stern, and Z. Sweedyk. The hardness of approximate optima in lattices, codes and linear equations. In *Proc. 34th IEEE Symp. on Foundations of Computer Science*, pages 724–733, 1993.
- CDKM00. Robert Carr, Srinivas Doddi, Goran Konjevod, and Madhav Marathe. On the red-blue set cover problem. In *Proc. 11th ACM-SIAM Symp. on Discrete Algorithms*, 2000.
- Chv79. V. Chvatal. A greedy heuristic for the set-covering problem. *Math. of Oper. Res.*, 4:233–235, 1979.
- DS99. I. Dinur and S. Safra. On the hardness of approximating label cover. Technical Report 15, Electronic Colloquium on Computational Complexity, 1999.
- EP00. Michael Elkin and David Peleg. The hardness of approximating spanner problems. In *Proc. 17th Symp. on Theoretical Aspects of Computer Science*, pages 370–381, February 2000.

- GM97. Michael H. Goldwasser and Rajeev Motwani. Intractability of assembly sequencing: Unit disks in the plane. In *Proc. Workshop on Algorithms and Data Structures*, volume LNCS 1272, pages 307–320. Springer-Verlag, 1997.
- Has00. Y. Hassin. Private communication, 2000.
- Hoc95. Dorit S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Co., Boston, MA, 1995.
- Kor98. G. Kortsarz. On the hardness of approximating spanners. In *Proc. 1st Int. Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, pages 135–146. Springer-Verlag, 1998. LNCS 1444.

# Approximation Algorithms for Maximum Linear Arrangement

Refael Hassin and Shlomi Rubinstein

Department of Statistics and Operations Research, School of Mathematical Sciences,  
Tel-Aviv University, Tel-Aviv 69978, Israel. {hassin,shlom}@math.tau.ac.il

**Abstract.** The GENERALIZED MAXIMUM LINEAR ARRANGEMENT PROBLEM is to compute for a given vector  $x \in \mathbb{R}^n$  and an  $n \times n$  non-negative symmetric matrix  $w = (w_{i,j})$ , a permutation  $\pi$  of  $\{1, \dots, n\}$  that maximizes  $\sum_{i,j} w_{\pi_i, \pi_j} |x_j - x_i|$ . We present a fast  $\frac{1}{3}$ -approximation algorithm for the problem. We also introduce a  $\frac{1}{2}$ -approximation algorithm for MAX  $k$ -CUT WITH GIVEN SIZES. This matches the bound obtained by Ageev and Sviridenko, but without using linear programming.

## 1 Introduction

We define the GENERALIZED LINEAR ARRANGEMENT PROBLEM as the problem of computing for a given vector  $x = (x_1 \leq \dots \leq x_n) \in \mathbb{R}^n$  of ‘points’ and an  $n \times n$  non-negative symmetric matrix  $w = (w_{i,j})$  of ‘weights’, a permutation  $\pi$  of  $\{1, \dots, n\}$  so that  $\sum_{i,j} w_{\pi_i, \pi_j} |x_j - x_i|$  is optimized. In an illustrative example, consider  $n$  linearly ordered points in which a set of  $n$  machines is to be located, and  $w_{i,j}$  is a measure of association of the  $i$ -th and  $j$ -th machines. Our interest is in the maximization version, the GENERALIZED MAXIMUM LINEAR ARRANGEMENT PROBLEM (GMLAP), where the goal is to maximize  $\sum_{i,j} w_{\pi_i, \pi_j} |x_j - x_i|$ , and keep the machines far from each other (compare with [7]).

The special (NP-hard) case in which  $x_i = i$  is known as the LINEAR ARRANGEMENT PROBLEM. Another special case of the problem is MAX CUT PROBLEM WITH GIVEN SIZES OF SIDES where for some  $p \leq \frac{n}{2}$ ,  $x_1 = \dots = x_p = 0$  and  $x_{p+1} = \dots = x_n = 1$ . Ageev and Sviridenko [3] applied a novel method of rounding linear programming relaxations and developed a  $\frac{1}{2}$ -approximation algorithm for this problem. ([2] contains a  $\frac{1}{2}$ -approximation for the more general directed version of the problem.) They also obtained a similar result for a more general MAX  $k$ -CUT PROBLEM in which integers  $p_1, \dots, p_k$  are given and the goal is to compute a  $k$ -cut, that is, a partition  $S_1, \dots, S_k$  of  $\{1, \dots, n\}$  with  $|S_i| = p_i$   $i = 1, \dots, k$ , which maximizes the weight of edges whose ends are in different sides of the partition.

The GMLAP is a special case of the MAXIMUM QUADRATIC ASSIGNMENT PROBLEM. In this problem two  $n \times n$  nonnegative symmetric matrices  $A = (a_{i,j})$  and  $B = (b_{i,j})$  are given and the objective is to compute a permutation  $\pi$  of  $\{1, \dots, n\}$  so that  $\sum_{\substack{i,j \in V \\ i \neq j}} a_{\pi(i), \pi(j)} b_{i,j}$  is maximized. A  $\frac{1}{4}$ -approximation algorithm for this problem, under the assumption that the values of one of the

matrices satisfy the triangle inequality, is given in [4]. Of course, this bound applies also to the GMLAP.

We will present  $\frac{1}{3}$ -approximation algorithms for the GMLAP. An interesting feature of our algorithm is that it simultaneously approximates the max cut problems with sizes  $p$  and  $n - p$  for all possible values of  $p$ . We also present an alternative  $\frac{1}{2}$ -approximation for MAX  $k$ -CUT PROBLEM WITH GIVEN SIZES OF THE SIDES. Unlike the algorithm of Ageev and Sviridenko, the latter algorithm doesn't use linear programming. The full version of this paper also contains a randomized  $\frac{1}{2}$ -approximation for the MAXIMUM LINEAR ARRANGEMENT PROBLEM.

We first describe, in Section 2, a generic randomized approximation algorithm for MAX CUT WITH GIVEN SIZES OF SIDES. The analysis of this special case will be used in Section 3 where we obtain our main result on the GMLAP. In Section 4 we treat the MAX  $k$ -CUT PROBLEM WITH GIVEN SIZES OF THE SIDES.

For a partition  $(S, T)$  we mean by  $(i, j) \in (S, T)$  that  $i \in S$  and  $j \in T$ . We denote by  $opt$  the optimal solution value in the problem under consideration.

## 2 Max Cut with Given Sizes of Sides

Given an undirected graph  $G = (V, E)$  with edge weights  $w_{i,j}$  ( $i, j \in E$ ) and  $|V| = n$ , a *cut* is a partition  $(S, T)$  of  $V$  and its weight is  $\sum_{s \in S} \sum_{t \in T} w_{s,t}$ . The problem is to compute a maximum weight cut such that  $|S| = p$ . Without loss of generality, we assume that  $p \leq \frac{n}{2}$ .

```

Max_Cut
input
  1. A graph  $G = (V, E)$   $V = \{1, \dots, n\}$  with edge weights  $w_{i,j}$  ( $i, j \in E$ ).
  2. An integer  $p \leq \frac{n}{2}$ .
returns
  A partition  $S, T$  of  $V = \{1, \dots, n\}$  such that  $|S| = p$ .
begin
  for  $i = 1, \dots, n$ 
     $W_i := \sum_{j:(i,j) \in E} w_{i,j}$ .
  end for
   $P := \{i_1, \dots, i_{2p} \in V \mid W_i \geq W_j \ \forall i \in P \ j \notin P\}$ .
  Randomly choose  $p$  nodes from  $P$  to form  $S$ .
   $T := V \setminus S$ .
  return  $S, T$ .
end Max_Cut

```

**Fig. 1.** Algorithm *Max\_Cut*

**Theorem 1** Let  $w(S, T)$  be the expected weight of the partition returned by Algorithm *Max\_Cut* (Figure 1). Then,  $w(S, T) \geq \frac{opt}{3}$ .

**Proof:** The probability that any given edge  $(i_j, l)$   $i_j \in P$  to be separated by  $(S, T)$  is  $\frac{1}{2}$  for both cases  $l \in P$  and  $l \notin P$ . Consider an optimal solution and denote by  $OPT$  the set of size  $p$  in it. Let  $r = |OPT \cap P|$ ,  $s = \sum_{(i,j) \in (OPT \cap P, P \setminus OPT)} w_{i,j}$ , and  $t = \sum_{(i,j) \in (OPT \cap P, V \setminus P)} w_{i,j}$ . Note that by definition of  $P$ , there is some threshold  $k$  such that  $W_i$  is at least  $k$  for  $i \in P$  and at most  $k$  for  $i \in OPT \setminus P$ . Also note that edges with two ends in  $P$  are counted twice in  $\sum_{i \in P} W_i$  so that

$$w(S, T) \geq \frac{s + \sum_{i \in P \setminus OPT} W_i}{4} + \frac{t}{2} \geq \frac{s + (2p - r)k}{4} + \frac{t}{2},$$

and

$$opt \leq (p - r)k + t + s.$$

We note that to compute a minimum possible value for the ratio  $\frac{w(S, T)}{opt}$  given that it can be made smaller than  $\frac{1}{2}$ , we can assume w.l.o.g. that  $t = 0$ . Let  $s = \alpha(2p - r)k$ . We now distinguish two cases.

Suppose first that  $\alpha \leq 1$ . We use

$$w(S, T) \geq k(1 + \alpha) \frac{2p - r}{4}$$

and

$$opt \leq k((2p - r)(1 + \alpha) - p),$$

so that

$$\frac{w(S, T)}{opt} \geq \frac{1}{4} \frac{(2p - r)(1 + \alpha)}{(2p - r)(1 + \alpha) - p}.$$

This ratio is monotone decreasing in  $\alpha$  so that for the worst case we substitute  $\alpha = 1$  and obtain

$$\frac{w(S, T)}{opt} \geq \frac{2p - r}{2(3p - 2r)}.$$

This expression is maximized when  $r = 0$ , in which case we obtain the ratio  $\frac{1}{3}$ .

Suppose now that  $\alpha > 1$ . We use the inequalities

$$w(S, T) \geq \frac{s}{2},$$

and

$$opt \leq (p - r)k + s \leq \frac{2p - r}{2}k + s \leq \frac{s}{2\alpha} + s \leq \frac{3}{2}s,$$

so that

$$\frac{w(S, T)}{opt} \geq \frac{1}{3}.$$

Algorithm *Max-Cut* is fast and simple, but for our results in the next section we will use a variation of it, which is also easier for derandomization. In this variation we change the main step of the algorithm as described in Figure 2

**Theorem 2** *Let  $w(S, T)$  be the expected weight of the partition returned by Max-Cut with the modification given in Figure 2. Then,  $w(S, T) \geq \frac{opt}{3}$ .*

**Proof:** As in Theorem 1 with  $P = \{1, \dots, 2p\}$ . ■

```

begin
Sort  $V$  in non-increasing order of  $W_i$ .
(For simplicity, suppose that  $W_1 \geq \dots \geq W_n$ .)
for  $i = 1, \dots, p$ 
    Assign  $2i - 1$  to  $S$ , with probability  $\frac{1}{2}$ . Assign  $2i$  to  $S$  otherwise.
end for

```

**Fig. 2.** Modified *Max\_Cut*

### 3 Generalized Maximum Linear Arrangement

We start by presenting an alternative way to compute the weight of a solution  $\pi$  to GMLAP (cf. [6]): For  $p = 1, \dots, n - 1$  let  $C_p = \sum_{i=1}^p \sum_{j=p+1}^n w_{\pi_i, \pi_j}$ . Note that the problem of maximizing  $C_p$  over all permutations  $\pi$  of  $\{1, \dots, n\}$  is the max cut problem with sizes of sides  $p$  and  $n - p$ . Now we observe that

$$\sum_{i,j} w_{\pi_i, \pi_j} |x_j - x_i| = \sum_{p=1}^{n-1} C_p |x_{p+1} - x_p|. \quad (1)$$

In other words, the contribution of the interval  $[x_p, x_{p+1}]$  to the weight of the solution is  $C_p |x_{p+1} - x_p|$ . Our algorithm *Randomized\_GMLA* (see Figure 3) approximates simultaneously all of these cut problems with factor  $\frac{1}{3}$  each, and consequently the same bound applies to the GMLAP instance as well.

*Randomized\_GMLA*

**input**

1. A non-negative symmetric matrix  $W = (w_{i,j} \ i, j = 1, \dots, n)$ .

2. A set of points  $x_1, \dots, x_n \in \mathbb{R}$ .

**returns** A permutation  $\pi_1, \dots, \pi_n$  of  $V = \{1, \dots, n\}$ .

**begin**

**for**  $i = 1, \dots, n$

$W_i := \sum_{j \in V \setminus \{i\}} w_{i,j}$ .

**end for**

*Sort  $V$  in non-increasing order of  $W_i$ .*

(For simplicity, suppose that  $W_1 \geq \dots \geq W_n$ .)

**for**  $i = 1, \dots, \lfloor \frac{n}{2} \rfloor$

Set  $\pi_i := 2i - 1$  and  $\pi_{n-i+1} := 2i$  with probability  $\frac{1}{2}$ .

Set  $\pi_i := 2i$  and  $\pi_{n-i+1} := 2i - 1$  otherwise.

If  $n$  is odd, set  $\pi_{\frac{n+1}{2}} := n$ .

**end for**

**return**  $\pi$ .

**end Randomized\_GMLA**

**Fig. 3.** Algorithm *Randomized\_GMLA*

**Theorem 3** Let  $\pi$  be the permutation returned by Randomized\_GMLA.

1. Let  $S_p = \{\pi_1, \dots, \pi_p\}$ ,  $T_p = \{\pi_{p+1}, \dots, \pi_n\}$ . Then  $(S_p, T_p)$  is a  $\frac{1}{3}$ -approximation for the max cut problem with sizes of sides  $p$  and  $n - p$ .
2.  $\pi$  is a  $\frac{1}{3}$ -approximation for the GMLAP.

**Proof:** By Theorem 2, for  $p = 1, \dots, n - 1$ , the value of  $C_p$  in the output of the algorithm is a  $\frac{1}{3}$ -approximation for the respective max cut problem. The proof for the second part of the theorem follows now from Equation II. ■

Derandomizing the algorithm is particularly simple. We apply the ‘method of conditional expectations’. Consider the  $i$ -th iteration of the algorithm. We should set  $\pi_i$  to either  $2i - 1$  or to  $2i$ , and  $\pi_{n-i+1}$  to the other value. This is done so that the expected value of the solution is maximized given the previous assignments and assuming that the following ones will be done according to Randomized\_GMLA. We call the resulting algorithm GMLA.

**Theorem 4** Let  $m = |\{(i, j) : w_{i,j} > 0\}|$ . Then, Algorithm GMLA computes a  $\frac{1}{3}$ -approximation for the GMLAP and for MAX CUT WITH GIVEN SIZES OF THE SIDES for every  $p = 1, \dots, \lfloor \frac{n}{2} \rfloor$  in time  $O(m + n \log n)$ .

## 4 Max $k$ -Cut with Given Sizes of the Sides

Given a graph  $G = (V, E)$  with edge weights  $w$  and integers  $p_1, \dots, p_k$  such that  $\sum p_i = n$ , the MAX  $k$ -CUT WITH GIVEN SIZES OF THE SIDES is to compute a  $k$ -cut, that is, a partition  $S_1, \dots, S_k$  of  $V$  such that  $|S_i| = p_i$   $i = 1, \dots, k$ , which maximizes the weight of edges whose ends are in different parts of the partition.

A vertex  $v \in V$  is said to *cover* the weight of the edges  $\{(u, v) \in E\}$ . A subset  $V' \subset V$  covers the weight of the union of edges which have at least one end in it. Bar-Yehuda [5] developed an  $O(n^2)$   $\frac{1}{2}$ -approximation algorithm for the following problem: Given  $w$ , compute a vertex set of minimum size that covers edge weight of size at least  $w$ .

One can obtain from this result, in a straightforward way, a solution to the following problem: Given  $p \leq \frac{1}{2}n$  find a set  $S'$  of  $2p$  vertices that covers edge weight of at least  $w(p)$ , where  $w(p)$  is the maximum edge weight that can be covered by  $p$  vertices. To achieve this goal we apply binary search over  $[0, w(E)]$ , where  $w(E)$  is the total weight of  $E$ . For each test value,  $w$ , we apply Bar-Yehuda’s algorithm and we stop with the highest value for which the algorithm returns a set  $S'$  with at most  $2p$  vertices. The complexity of this procedure is  $O(n^2 \log w(E))$ .

Our algorithm for the case of  $k = 2$  (MAX CUT WITH GIVEN SIZES OF THE SIDES) proceeds as follows: Randomly select  $p$  vertices from  $S'$  and move them to the other side of the cut. Let the resulting set be  $S_A$ . We claim that the expected size of the cut  $(S_A, V \setminus S_A)$  is a  $\frac{1}{2}$ -approximation for the problem. The argument is that the weight of the edges covered by  $S'$  is an upper bound on the optimal solution value, and each of these edges will be in the cut with probability  $\frac{1}{2}$ . The algorithm can be derandomized by applying the ‘method of conditional expectations’. Alternatively, the rounding method of Ageev and Sviridenko can

also be used to obtain a  $\frac{1}{2}$ -approximation in deterministic linear time, once  $S'$  is given  $\blacksquare$ .

Our algorithm can be modified for the MAX  $k$ -CUT PROBLEM WITH GIVEN SIZES OF THE SIDES as well: We first observe that if  $p_i \leq \frac{1}{2}n$  for every  $i = 1, \dots, k$  then the cut contains more than half of the edges so that a random solution has expected weight of at least half the total weight of the graph. Thus a random solution suffices to obtain a  $\frac{1}{2}$ -approximation.

Assume now that  $p_1 > \frac{n}{2}$ . We compute as above sets  $S'$  and  $S_A$  with  $p = n - p_1$ . We set  $P_1 = V \setminus S_A$  and arbitrarily partition  $S_A$  to form  $P_2, \dots, P_k$ . The resulting  $k$ -cut has the property that the expected weight of edges between  $P_1$  and the other parts is already half the weight of the edges covered by  $S'$  which is itself an upper bound on the optimal solution. Thus the  $k$ -cut we constructed is a  $\frac{1}{2}$ -approximation for the problem. Again, the algorithm can be derandomized.

## References

1. A. A. Ageev, personal communication.
2. A. A. Ageev, R. Hassin, and M. I. Sviridenko, “Directed max cut with given sizes of parts”, 1999.
3. A. A. Ageev and M. I. Sviridenko, “Approximation algorithms for maximum coverage and max cut with given sizes of parts”, *IPCO’99*.
4. E. Arkin, R. Hassin and M. I. Sviridenko, “Approximating the maximum quadratic assignment problem”. A preliminary version appeared in *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA2000)*, 889-890.
5. R. Bar-Yehuda, “Using homogeneous weights for approximating the partial cover problem”, *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA99)*, 71-75, 1999.
6. J-C. Picard and H. D. Ratliff, “A cut approach to the rectilinear distance facility location problem”, *Operations Research* **26**, 1978, 422-433.
7. A. Tamir, “Obnoxious facility location on graphs,” *SIAM J. Discrete Mathematics* **4**, 1991, 550-567.

# Approximation Algorithms for Clustering to Minimize the Sum of Diameters<sup>1</sup>

Srinivas R. Doddi<sup>1</sup>, Madhav V. Marathe<sup>2</sup>, S. S. Ravi<sup>3</sup>,  
David Scot Taylor<sup>4</sup>, and Peter Widmayer<sup>5</sup>

<sup>1</sup> Los Alamos National Laboratory, P. O. Box 1663, MS B265  
Los Alamos, NM 87545, USA  
[srinu@lanl.gov](mailto:srinu@lanl.gov)

<sup>2</sup> Los Alamos National Laboratory, P. O. Box 1663, MS M997  
Los Alamos, NM 87545, USA  
[marathe@lanl.gov](mailto:marathe@lanl.gov)

<sup>3</sup> Department of Computer Science  
University at Albany - State University of New York  
Albany, NY 12222, USA  
[ravi@cs.albany.edu](mailto:ravi@cs.albany.edu)

<sup>4</sup> Department of Computer Science  
University of California  
Los Angeles, CA 90095-1596, USA  
[dstaylor@cs.ucla.edu](mailto:dstaylor@cs.ucla.edu)

<sup>5</sup> Institute for Theoretical Computer Science  
ETH, 8092 Zürich, Switzerland  
[widmayer@inf.ethz.ch](mailto:widmayer@inf.ethz.ch)

**Abstract.** We consider the problem of partitioning the nodes of a complete edge weighted graph into  $k$  clusters so as to minimize the sum of the diameters of the clusters. Since the problem is NP-complete, our focus is on the development of good approximation algorithms. When edge weights satisfy the triangle inequality, we present the first approximation algorithm for the problem. The approximation algorithm yields a solution that has no more than  $10k$  clusters such that the total diameter of these clusters is within a factor  $O(\log(n/k))$  of the optimal value for  $k$  clusters, where  $n$  is the number of nodes in the complete graph. For any fixed  $k$ , we present an approximation algorithm that produces  $k$  clusters whose total diameter is at most twice the optimal value. When the distances are not required to satisfy the triangle inequality, we show that, unless  $P = NP$ , for any  $\rho \geq 1$ , there is no polynomial time approximation algorithm that can provide a performance guarantee of  $\rho$  even when the number of clusters is fixed at 3. Other results obtained include a polynomial time algorithm for the problem when the underlying graph is a tree with edge weights.

---

<sup>1</sup> Research Supported by Department of Energy Contract W-7405-ENG-36 and by NSF Grant CCR-97-34936.

# 1 Introduction

## 1.1 Motivation

The main goal of clustering is to partition a set of objects into homogeneous and well separated subsets (clusters). Clustering techniques have been used in a wide variety of application areas including information retrieval, image processing, pattern recognition and database systems [Ra97, ZRL96, JD88, DH73]. Over the last three decades, several clustering methods have been developed for specific applications [HJ97, JD88]. Many of these methods define a distance (or a similarity measure) between each pair of objects, and partition the collection into clusters so as to optimize a suitable objective based on the distances. Some of the objectives that have been studied in the literature include minimizing the maximum diameter or radius, total pairwise distances in clusters, etc. The survey paper by Hansen and Jaumard [HJ97] provides an extensive list of clustering objectives and applications for these objectives.

Clustering problems where the objective is to minimize the maximum cluster diameter have been well studied from an algorithmic point of view (see Section 1.4 for a summary). The focus of this paper is on clustering problems where the objective is to partition a given collection of objects into a specified number of clusters so as to minimize the sum of the diameters of individual clusters. The motivation for this objective is derived from the fact that in several applications, clustering algorithms that minimize the maximum diameter produce a “dissection effect” [HJ97, MS89]. This effect causes objects that should normally belong to the same cluster to be assigned to different clusters, as otherwise the diameter of a cluster becomes too large. In such applications, the sum of diameters objective is more useful as it reduces the dissection effect [HJ97, MS89].

## 1.2 Problem Formulation and Previous Work

To study the clustering problem in a general setting, we represent the objects to be clustered as nodes of a complete edge-weighted undirected graph  $G(V, E)$  with  $|V| = n$ . The distance (or similarity measure) between any pair of objects can then be represented as the weight of the corresponding edge in  $E$ . For an edge  $\{u, v\}$  in  $E$ , we use  $\omega(u, v)$  to denote the weight of the edge. It is assumed that the edge weights are nonnegative. For any subset  $V'$  of  $V$ , the **diameter** of  $V'$  (denoted by  $\text{DIA}(V')$ ) is the weight of a largest edge in the complete subgraph of  $G$  induced on  $V'$ . Note that when  $|V'| = 1$ ,  $\text{DIA}(V') = 0$ . A formal statement of the clustering problem considered in this paper is as follows.

### Clustering to Minimize Sum of Diameters (CMSD)

**Instance:** A complete graph  $G(V, E)$ , a nonnegative weight (or distance)  $\omega(u, v)$  for each edge  $\{u, v\}$  in  $E$  and an integer  $k \leq |V|$ .

**Requirement:** Partition  $V$  into  $k$  subsets  $V_1, V_2, \dots, V_k$  such that  $\sum_{i=1}^k \text{DIA}(V_i)$  is minimized.

In general, edge weights in instances of CMSD need not satisfy the triangle inequality. We use  $\text{CMSD}_\Delta$  to denote instances of CMSD where edge weights satisfy the triangle inequality. Most of our results are for the  $\text{CMSD}_\Delta$  problem. We assume without loss of generality that the optimal solution value to any given instance of  $\text{CMSD}_\Delta$  is strictly greater than zero. We may do so since it is easy to determine whether a given instance of  $\text{CMSD}_\Delta$  can be partitioned into a specified number of clusters each of which has a diameter of zero.

We now summarize the known results from the algorithmic literature for the CMSD problem. Brucker [Br78] showed that CMSD (without triangle inequality) is NP-complete for any fixed  $k \geq 3$ . Hansen and Jaumard [HJ87] studied the CMSD problem with  $k = 2$  and presented an algorithm with a running time of  $O(n^3 \log n)$ . They also showed that for  $k = 2$ , the minimization problem for any given function of the two diameters can be solved in  $O(n^5)$  time. When the input is specified as an undirected edge weighted graph with  $n$  nodes and  $m$  edges, Monma and Suri [MS89] showed that the CMSD problem for  $k = 2$  can be solved in time  $O(nm \log n)$ . This is an improvement over the algorithm of [HJ87] for sparse graphs. Brucker [Br78] observed that the 1-dimensional version of  $\text{CMSD}_\Delta$  can be solved efficiently for any value of  $k$ . For the Euclidean version of  $\text{CMSD}_\Delta$  with  $k = 2$ , Monma and Suri [MS89] presented an algorithm which uses  $O(n)$  space and runs in  $O(n^2)$  time. Capoyleas et al. [CRW91] also studied a generalized version of the  $\text{CMSD}_\Delta$  problem for points in  $\mathbb{R}^2$ . They showed that for any fixed  $k$ , the problem can be solved in polynomial time for any monotonic increasing function of cluster radius or diameter. Examples of such monotonic increasing functions include sum of diameters (or radii), maximum diameter (or radius), etc.

### 1.3 Summary of Main Results

We study the complexity and approximability of the CMSD problem. The main results of this paper can be summarized as follows:

1. We show that unless  $P = NP$ , CMSD cannot be efficiently approximated to within any factor even when the number of clusters is fixed at 3. (In contrast, note that CMSD is known to be efficiently solvable when the number of clusters is equal to 2 [HJ87, MS89].)
2. For  $\text{CMSD}_\Delta$ , we show that if the constraint on the number of clusters must be met, then it is NP-hard to approximate the total diameter to within a factor  $2 - \epsilon$ , for any  $\epsilon > 0$ .
3. In contrast to the non-approximability results above, we present a polynomial time bicriteria approximation algorithm [MR+98] for  $\text{CMSD}_\Delta$ . This approximation algorithm outputs a solution with at most  $10k$  clusters whose total diameter is within a factor of  $O(\log(n/k))$  of the minimum possible total diameter with  $k$  clusters.
4. We also show that when the number of clusters  $k$  is fixed, there is an approximation algorithm for  $\text{CMSD}_\Delta$  which produces at most  $k$  clusters whose total diameter is within a factor of 2 of the minimum possible total diameter.

A brief summary of our other results is given in Section 5.

## 1.4 Other Related Work

A number of researchers have addressed the clustering problem where the goal is to minimize the maximum diameter or radius of a cluster. In the location theory literature, the problem of minimizing the maximum radius is also known as the  $k$ -center problem. For the metric version of the problem of minimizing the maximum diameter, Gonzalez [Go85] presented a simple greedy heuristic that runs in  $O(nk)$  time and provides a performance guarantee of 2. He also showed that, unless  $P = NP$ , the performance guarantee cannot be improved. Using a general technique for approximating bottleneck problems, Hochbaum and Shmoys [HS86] also presented a heuristic with a performance guarantee of 2 for the metric version of the  $k$ -center problem.

In [FPT81, MS84], it is shown that the problems of minimizing the maximum radius or diameter remain NP-hard even for points in  $\mathbb{R}^2$ . For this geometric version, Feder and Greene [FG88] improved the running time of Gonzalez's heuristic to  $O(n \log n)$ . They also showed that it is NP-hard to achieve a performance guarantee of 1.82 and 1.97 respectively for the diameter and radius problems in  $\mathbb{R}^2$ . Recently, Agarwal and Procopiuc [AP98] have presented an exact algorithm with a running time of  $n^{O(k^{1-1/d})}$  for the  $k$ -center problem for points in  $\mathbb{R}^d$ . For any  $\epsilon > 0$ , they have also presented an  $(1 + \epsilon)$  approximation algorithm with a running time of  $O(n \log k) + (k/\epsilon)^{O(k^{1-1/d})}$  for the problem.

Plesník [Pl82] has addressed the problem of partitioning the *edges* of a given graph  $G(V, E)$  into  $k$  subsets so that each subset forms a connected graph on the vertex set  $V$ , and a given function of the diameters of the resulting subgraphs is minimized. The objectives considered in [Pl82] include minimizing the maximum diameter and minimizing the total diameter. It is shown that, unless  $P = NP$ , even for  $k = 2$ , these objectives cannot be efficiently approximated to within factors less than  $3/2$  and  $5/4$  respectively.

Several other types of clustering problems have also been studied in the literature. For example, Charikar et al. [CC+97] study an incremental version of the clustering problem for minimizing the maximum radius. Pferschy et al. [PRW94] study geometric versions of clustering problems using objectives such as minimizing the total perimeter. Agarwal and Procopiuc [AP00] study projective clustering problems where the goal is to cover a set of points in  $\mathbb{R}^d$  using hyper-strips, and the objective is to minimize the maximum width of the strips. References where other types of clustering problems are studied include [Ma99, ABC+98, GH98, DKS97, Da94, BKK94].

## 2 Preliminaries

### 2.1 A Simple Upper Bound on the Optimal Solution Value

Given any instance of CMSD, we can easily construct a feasible solution consisting of  $k$  clusters with total diameter at most the maximum edge weight: form one cluster consisting of  $n - k + 1$  arbitrarily chosen vertices and make each of the

remaining  $k - 1$  vertices a singleton cluster. This observation is stated formally below.

*Remark 1.* For any instance of  $I$  of CMSD, the optimal solution value is at most the maximum edge weight in  $I$ .  $\square$

## 2.2 A Merging Lemma

The formulation of the CMSD problem requires that the clusters be pairwise disjoint. Our approximation algorithms may produce clusters which may not satisfy the disjointness condition. The following lemma points out that for instances of  $\text{CMSD}_\Delta$ , we can merge pairs of intersecting sets without increasing the total diameter.

**Lemma 1.** *Let  $I$  be an instance of  $\text{CMSD}_\Delta$  given by the edge weighted complete graph  $G(V, E)$  and integer  $k$ . Let  $\mathcal{C} = \{C_1, C_2, \dots, C_k\}$  be a collection of subsets of  $V$  such that their union is  $V$  and the sum of the diameters of all the subsets in  $\mathcal{C}$  is  $\psi$ . Further, suppose  $C_i$  and  $C_j$  ( $i \neq j$ ) are two sets in  $\mathcal{C}$  such that  $C_i \cap C_j \neq \emptyset$ . Then the total diameter of the collection  $\mathcal{C}'$  obtained by deleting  $C_i$  and  $C_j$  from  $\mathcal{C}$  and adding the set  $C_i \cup C_j$  is at most  $\psi$ .*

*Proof.* The lemma would follow by showing that  $\text{DIA}(C_i \cup C_j) \leq \text{DIA}(C_i) + \text{DIA}(C_j)$ . To do this, let  $x$  be a node in  $C_i \cap C_j$  and let  $u$  and  $v$  be two nodes in  $C_i \cup C_j$  such that  $\omega(u, v) = \text{DIA}(C_i \cup C_j)$ . If  $u$  and  $v$  are both in  $C_i$  (or both in  $C_j$ ), then  $\omega(u, v) \leq \text{DIA}(C_i)$  ( $\omega(u, v) \leq \text{DIA}(C_j)$ ), and the proof is trivial. So, assume that  $u \in C_i$  and  $v \in C_j$ . By the triangle inequality,  $\omega(u, v) \leq \omega(u, x) + \omega(v, x)$ . Since  $u$  and  $x$  are both in  $C_i$ ,  $\omega(u, x) \leq \text{DIA}(C_i)$ . Similarly,  $\omega(v, x) \leq \text{DIA}(C_j)$ . Therefore,  $\text{DIA}(C_i \cup C_j) = \omega(u, v) \leq \text{DIA}(C_i) + \text{DIA}(C_j)$ , and this completes the proof.  $\square$

In view of the above lemma, when considering instances of  $\text{CMSD}_\Delta$ , we may repeatedly merge pairs of clusters with nonempty intersection until the clusters are pairwise disjoint. The merging process does not increase the total diameter of the clusters.

## 2.3 Transformation to Weighted Set Cover

Our results rely on a transformation from instances of  $\text{CMSD}_\Delta$  to instances of the weighted set cover problem. Given an instance of  $\text{CMSD}_\Delta$  along with a nonnegative value  $f$ , the transformation in Figure 11 produces an instance of the weighted set cover problem. It is clear that the transformation can be carried out in polynomial time. The following lemma points out an important property of the resulting set cover instance.

**Lemma 2.** *Let  $I$  denote an instance of  $\text{CMSD}_\Delta$  problem and let  $f$  be a non-negative number. Let  $I'$  denote the instance of the weighted set cover problem produced by the transformation in Figure 11 when  $I$  and  $f$  are given as inputs. Let  $\text{OPT}(I)$  and  $\text{OPT}(I')$  denote the optimum solution values to  $I$  and  $I'$  respectively. Then,  $\text{OPT}(I') \leq 2\text{OPT}(I) + f$ .*

```

TRANSFORMTOSETCOVER( $G(V, E), k, f$ )
 $f$  is a nonnegative parameter.

```

**Output:** An instance of weighted set cover problem with base set  $Q$ , and collection  $\mathcal{W}$  of nonempty subsets of  $Q$ , each with a weight. The weight of a set  $W \in \mathcal{W}$  is denoted by  $c(W)$ .

1.  $Q = V$  /\* Note:  $|Q| = n$ . \*/
2.  $\mathcal{W} = \emptyset$
3. **for** each  $v \in V$  **do**
  - (a) Sort  $\{\omega(v, u) : u \in V\}$  into (strictly) increasing order.
  - (b) Let  $\alpha_1 = 0 < \alpha_2 < \dots < \alpha_{r_v}$  denote the sorted order.
  - (c) **for**  $i = 1$  to  $r_v$  **do**
    - i. Let  $W_v^i = \{u : \omega(u, v) \leq \alpha_i\}$
    - ii. Let  $c(W_v^i) = \text{DIA}(W_v^i) + f/k$
    - iii. Add  $(W_v^i)$  to  $\mathcal{W}$
4. **return** $(Q, \mathcal{W})$

**Fig. 1.** Transformation from CMSD $_{\Delta}$  to Weighted Set Cover

*Proof.* Let  $C_1, C_2, \dots, C_k$  denote the clusters in an optimal solution to  $I$ . Thus,  $\text{OPT}(I) = \sum_{i=1}^k \text{DIA}(C_i)$ . We will show that there is a subcollection of  $k$  sets in  $I'$  such that the sets in the subcollection together cover the base set  $Q$  and the total weight of the sets in the subcollection is at most  $2 \text{OPT}(I) + f$ . The lemma would then follow immediately.

Consider each cluster  $C_i$  ( $1 \leq i \leq k$ ) in the optimal solution to  $I$ . If  $C_i$  contains two or more nodes, let  $v_i$  be a node in  $C_i$  such that  $v_i$  is one of the endpoints of an edge whose weight is equal to  $\text{DIA}(C_i)$ . If  $C_i$  contains only one node (i.e.,  $\text{DIA}(C_i) = 0$ ), let  $v_i$  be that node. Now, by the transformation of Figure  $\square$ ,  $I'$  has a set, say  $W_i$ , that includes all the nodes which are at a distance of at most  $\text{DIA}(C_i)$  from  $v_i$ . By the triangle inequality,  $\text{DIA}(W_i) \leq 2 \text{DIA}(C_i)$ . So,  $c(W_i) = \text{DIA}(W_i) + f/k \leq 2 \text{DIA}(C_i) + f/k$ . Clearly, the subcollection  $\{W_1, W_2, \dots, W_k\}$  covers the base set  $Q$ . The weight of this cover is  $\sum_{i=1}^k c(W_i)$ , which is at most  $\sum_{i=1}^k (2 \text{DIA}(C_i) + f/k) = 2 \text{OPT}(I) + f$ . This completes the proof of the lemma.  $\square$

## 2.4 The Budgeted Maximum Coverage Problem

For obtaining our approximation result for CMSD $_{\Delta}$  (where the number of clusters  $k$  is a part of the problem instance), we use a known approximation result for the Budgeted Maximum Coverage Problem (BMCP). Below, we provide a definition of the problem and state the necessary approximation result.

An instance of BMCP consists of a base set  $Q = \{q_1, q_2, \dots, q_n\}$ , a collection  $\mathcal{W}$  of nonempty subsets of  $Q$ , a nonnegative weight  $c(W)$  for each set  $W \in \mathcal{W}$  and a nonnegative budget  $B$ . The goal is to choose a subcollection of sets from  $\mathcal{W}$

so that the total cost of the chosen sets is at most  $B$  and the number of elements covered by the chosen sets is maximum. This problem is NP-hard since it is a restatement of the minimum cost set cover problem. The following approximation result for BMCP is proved in [KMN99].

**Theorem 1.** BMCP can be efficiently approximated to within a factor  $(1 - 1/e)$ .  $\square$

It is shown in [KMN99] that the approximation algorithm referred to in Theorem 1 can also be used for the more general version of BMCP where there is a weight associated with each element of the base set, and the goal is to maximize the weight of the elements covered by the chosen sets. For our results, the unit weight version of BMCP where the weight of each element of the base set is 1, suffices.

### 3 Approximating CMSD $_{\Delta}$

#### 3.1 Algorithm Overview

We give a brief top-down description of our approximation algorithm APPROX-CMSD $_{\Delta}$ , and introduce the terminology used in the analysis. At all times, APPROX-CMSD $_{\Delta}$  maintains a set  $\mathcal{D}$  of clusters which cover all vertices in  $V$ , at cost  $\Psi$ . We call these *global clusters*, since they cover all vertices in  $V$ . The algorithm begins with  $\mathcal{D}$  consisting of  $|V|$  singleton clusters, and progresses through a series of rounds. During each round, it constructs a vertex set  $N$  by selecting an arbitrary vertex from each of its current clusters. It then finds a clustering  $\mathcal{C}$  on  $N$ . We call the clusters in  $\mathcal{C}$  *local clusters*, since they do not need to cover all of  $V$ , but only  $N$ . As will be shown, the number of clusters  $|\mathcal{C}|$  is at most  $3k[1 + \ln(|N|/k)]$ . We use  $\psi$  to denote their total cost. Next, APPROX-CMSD $_{\Delta}$  uses MERGE to suitably combine the clusters in  $\mathcal{D}$  into a set of just  $|\mathcal{C}|$  clusters, which cover all of  $V$  at cost at most  $\Psi + \psi$ . This entire process is repeated until the number of clusters in  $\mathcal{D}$  is at most  $10k$ .

APPROX-CMSD $_{\Delta}$  uses FINDCOVER to return the required  $\mathcal{C}$  clusters during each round. FINDCOVER, in turn, iterates through at most  $O(\ln(|N|/k))$  calls to PARAMETRICBMCP each of which returns a set of at most  $3k$  clusters which cover all but a  $1/e$  fraction of the remaining uncovered vertices from  $N$ . These clusters have cost no more than  $3(1 + \epsilon)OPT$ .

Using TRANSFORMTOSETCOVER from Figure 1, PARAMETRICBMCP converts the problem to a set cover instance, and repeatedly calls the Budgeted Maximum Coverage Approximation Algorithm BMCP, with growing budgets, until the budget is large enough to make BMCP cover the required fraction of vertices. A complete description of the approximation algorithm is given in Figure 2.

APPROX-CMSD $_{\Delta}$ ( $G(V, E), k$ )

**Output:** A set of no more than  $10k$  clusters with sum of diameters  $O(\ln(|V|/k)) \text{OPT}$ .

1.  $\mathcal{D} = \{\{v\} : v \in V\}$
2. **while** ( $|\mathcal{D}| > 10k$ ) **do** /\* We call each iteration a Round \*/
  - (a)  $N =$  Set of vertices obtained by choosing one arbitrary vertex from each  $D \in \mathcal{D}$ .
  - (b)  $\mathcal{C} = \text{FINDCOVER}(G(N, E'), k)$  /\*  $G(N, E')$  : Complete subgraph on  $N$  \*/
    - (c)  $\mathcal{D} = \text{MERGE}(\mathcal{D}, \mathcal{C})$
3. **return**( $\mathcal{D}$ )

FINDCOVER( $G(N, E), k$ )

**Output:** A set of no more than  $3k[1 + \ln(|N|/k)]$  clusters which cover  $N$  with cost no more than  $3[1 + \ln(|N|/k)](1 + \epsilon)\text{OPT}$ .

1.  $\mathcal{C} = \emptyset$
2. **while** ( $N \neq \emptyset$ ) **do**
  - (a)  $\mathcal{C}' = \text{PARAMETRICBMCP}(G(N, E), k)$
  - (b)  $\mathcal{C} = \mathcal{C} \cup \mathcal{C}'$
  - (c)  $N = N - \{i : i \in C \text{ for some } C \in \mathcal{C}'\}$
  - (d)  $E =$  Edges in the complete subgraph induced on the new, smaller  $N$
3. **return**( $\mathcal{C}$ )

PARAMETRICBMCP( $G(N, E), k$ )

**Output:** A set of no more than  $3k$  clusters which cover  $(1 - 1/e)|N|$  or more vertices from  $N$  with cost no more than  $3(1 + \epsilon)\text{OPT}$  for any fixed  $\epsilon > 0$ .

1.  $f =$  the smallest non-zero edge weight in  $G(N, E)$
2.  $\mathcal{C}' = \{\{v\} : v \in N\}$
3. **while** ( $|\mathcal{C}'| > 3k$  **or**  $|\{v : v \in C \text{ for some } C \in \mathcal{C}'\}| < (1 - 1/e)|N|$ ) **do**
  - (a)  $\mathcal{S} = \text{TRANSFORMTOSETCOVER}(G(N, E), k, f)$
  - (b)  $\mathcal{C}' = \text{BMCP}(\mathcal{S}, 3f)$
  - (c)  $f = (1 + \epsilon)f$
4. **return**( $\mathcal{C}'$ )

MERGE( $\mathcal{D}, \mathcal{C}$ )

**Remark:**  $\mathcal{D}, \mathcal{C}$  collections of vertex sets such that  $\forall D \in \mathcal{D}, \exists C \in \mathcal{C}$  such that  $(D \cap C \neq \emptyset)$ .

**Output:** A set of  $|\mathcal{C}|$  vertex sets which cover all  $\{v : v \in X \text{ for some } X \in \mathcal{D} \cup \mathcal{C}\}$  at cost no more than the sum of the costs of  $\mathcal{C}$  and  $\mathcal{D}$ .

1. **for** each  $C \in \mathcal{C}$  **do**
  - for** each  $D \in \mathcal{D}$  **do**
    - if** ( $C \cap D \neq \emptyset$ ) **then**

$$C = C \cup D ; \mathcal{D} = \mathcal{D} - D$$
2. **return**( $\mathcal{C}$ )

**Fig. 2.** Outline of APPROX-CMSD $_{\Delta}$

### 3.2 Correctness of Algorithm

To show that our algorithm runs in polynomial time and achieves the stated performance guarantees, we analyze it from the lower level functions up to the top level call, beginning with PARAMETRICBMCP, and finishing with APPROX-CMSD $_{\Delta}$ .

**Lemma 3.** *Given graph  $G$  with optimal  $k$ -cluster cost  $\text{OPT}$ , PARAMETRICBMCP returns no more than  $3k$  clusters which contain at least  $(1 - 1/e)|N|$  of the vertices from  $|N|$ . Further, the sum of diameters of the returned clusters is no more than  $3(1 + \epsilon)\text{OPT}$ .*

*Proof.* By Lemma 2, the call to TRANSFORMTOSETCOVER returns a set cover instance with optimal solution no more than  $2\text{OPT} + f$ . When  $f > \text{OPT}$ , by Theorem 1, the call to BMCP with budget  $3f > 3\text{OPT} > 2\text{OPT} + f$ , will return sets which cover the stated number of vertices. Also, when  $f > 0$ , this solution cannot have more than  $3k$  clusters: each of the clusters has minimum cost  $f/k$ , so any more than  $3k$  clusters will have cost more than  $3f$ . Therefore, with any  $f > \text{OPT}$ , the call to BMCP with budget  $3f$  will return at most  $3k$  clusters which cover enough vertices.

Since we start  $f$  at the smallest possible (non-zero) value (in fact, we first implicitly test if  $f = 0$  suffices), and increase it by factors of  $(1 + \epsilon)$ , we are guaranteed to try a value  $f$  such that  $f < (1 + \epsilon)\text{OPT}$ . This will occur within  $O(\log_{1+\epsilon} \text{OPT})$  iterations. Since  $\text{OPT}$  is at most the maximum edge weight (Remark 1), the number of iterations is polynomial.  $\square$

**Lemma 4.** *Given graph  $G(N, E)$  with optimal  $k$ -cluster cost  $\text{OPT}$ , FINDCOVER returns no more than  $3k[1 + \ln(|N|/k)]$  clusters which cover  $N$  with cost no more than  $3[1 + \ln(|N|/k)](1 + \epsilon)\text{OPT}$ .*

*Proof.* By Lemma 3, each call to PARAMETRICBMCP will return at most  $3k$  clusters of cost  $3(1 + \epsilon)\text{OPT}$ , and will leave at most  $|N|/e$  of the  $|N|$  vertices uncovered. In the ensuing iterations of PARAMETRICBMCP, we use a subset of  $N$  which certainly has an optimal  $k$ -clustering with cost no greater than  $\text{OPT}$ . After  $i$  iterations, we are guaranteed to have no more than  $3k$  remaining vertices, where  $|N|/e^i \leq 3k$ . To upper bound  $i$ , notice that if  $i$  is not the last iteration,  $|N|/e^{i-1} > 3k$ , and  $i \leq 1 + \ln(|N|/3k) \leq \ln(|N|/k)$ . The final iteration generates at most  $3k$  additional singleton clusters with cost zero. Each of the  $1 + \ln(|N|/k)$  iterations returns no more than  $3k$  clusters, of cost at most  $3(1 + \epsilon)\text{OPT}$ . The lemma follows.  $\square$

**Lemma 5.** *MERGE returns  $|\mathcal{C}|$  vertex sets which cover all  $\{v : v \in X \text{ for some cluster } X \in \mathcal{D} \cup \mathcal{C}\}$ , with cost no more than the sum of the costs of  $\mathcal{C}$  and  $\mathcal{D}$ .*

*Proof.* Consider all  $\mathcal{C} \cup \mathcal{D}$  clusters whose cost is the sum of the costs of  $\mathcal{C}$  and  $\mathcal{D}$ . Since each  $D \in \mathcal{D}$  intersects some  $C \in \mathcal{C}$ , we may replace  $D$  and  $C$  with  $D \cup C$ , at no additional cost, by Lemma 1. This process can be continued until each cluster in  $\mathcal{D}$  has been merged into some cluster in  $\mathcal{C}$ .  $\square$

Finally, we need to show that the top level function APPROX-CMSD $_{\Delta}$  does in fact halt within a polynomial number of iterations. To do this, we show that the number of clusters in  $\mathcal{D}$  is eventually less than  $10k$ , and that this happens after no more than  $O(\log_2 \log_2 (n/k))$  rounds.

Our algorithm begins with  $n$  vertices, and by Lemma 4, after the end of the first round, we are left with  $3k[1 + \ln(n/k)]$  clusters, each of which contributes one vertex towards the second round. Generalizing this for all rounds, let  $\mathcal{D}_i$  be the set of global clusters at the end of round  $i$ , and  $n_i = |\mathcal{D}_i|$ . Then,  $n_0 = n$ , and  $n_{i-1}$  is both the number of clusters at the end of round  $i - 1$  and the number of vertices we need to cluster in the  $i^{\text{th}}$  round. We get the recurrence

$$n_{i+1} \leq 3k[1 + \ln(n_i/k)].$$

Let  $t_i = n_i/k$ , we have  $t_{i+1} \leq 3 + 3 \cdot \ln t_i \leq 6 \cdot \ln t_i$  for  $t_i \geq e$ . By having enough rounds to make  $t_i$  constant, we will have a total of  $O(k)$  clusters. After  $O(\log^* t_0)$  rounds,  $t_i$  becomes a constant, but here we will instead give a simple proof that  $O(\log_2 \log_2 t_0) = O(\log_2 \log_2 (n/k))$  rounds are sufficient.

**Lemma 6.** *After at most  $5 + \log_2 \log_2 (n/k)$  rounds,  $|\mathcal{D}|$  contains at most  $10k$  clusters.*

*Proof.* Consider the “iterating” function used to get  $\log^* x$  from  $\log_2 x$ . For any function  $f$  such that  $f(x) < x$  for sufficiently large  $x$ , the iterating function is the number of times you must apply that function to get a constant. More specifically, define the function  $(f)^*(x)_C$  to be the number of times that  $f()$  must be iteratively applied to get a result less than  $C$ . (Thus,  $(\log_2)^*(x)_1$  gives the familiar function  $\log^* x$ .) Next, we use the fact that for  $x > 2109$ ,  $6 \cdot \ln x < \sqrt{x}$ . Thus,  $(6 \cdot \ln)^*(x)_{2109} \leq (\sqrt{\cdot})^*(x)_{2109} \leq (\sqrt{\cdot})^*(x)_1$ . However,  $(\sqrt{\cdot})^*(x)_1 = \lceil \log_2 \log_2 x \rceil$ , so we need to iterate less than  $\log_2 \log_2 t_0$  times before reaching  $t_i \leq 2109$ . One more iteration for  $n$  gives us  $n_{1+\log_2 \log_2 t_1} \leq 3k + 3k \cdot \ln 2109 \leq 26k$ . Applying the recursion four more times gives  $n_{5+\log_2 \log_2 (n/k)} \leq 10k$ .  $\square$

Thus, APPROX-CMSD $_{\Delta}$  will terminate in  $O(\log \log (n/k))$  rounds. Each round has a call to FINDCOVER, which makes at most  $O(\log (n/k))$  calls to PARAMETRICBMCP. Using  $T(x)$  to denote the running time of BMCP, the time taken by all the calls to PARAMETRICBMCP is  $O((n^2 \log n + T(n^2)) \log_{1+\epsilon} \text{OPT})$ . Thus, the running time of the approximation algorithm is

$$O(\log \log (n/k)[\log (n/k)(n^2 \log n + T(n^2)) \log_{1+\epsilon} \text{OPT}]).$$

Since  $T(x)$  is polynomial by [KMN99], so is our algorithm.

Now all that is left is to show that the total cost is no more than the stated bound. Let  $\mathcal{C}_i$  denote the set of local clusters from round  $i$ . Since  $\mathcal{C}_i$  covers the set of  $N$  vertices, one from each  $D \in \mathcal{D}_i$ , we know that each  $D \in \mathcal{D}_i$  intersects a cluster  $C$  in  $\mathcal{C}_i$ . Let  $\Psi_i$  and  $\psi_i$  be the sum of diameters of the global and local clusters during the  $i^{\text{th}}$  round respectively. The following lemma can be proven by induction on  $i$ .

**Lemma 7.** After  $i$  rounds of APPROX-CMSD $_{\Delta}$ ,  $\Psi_i \leq \sum_{j=1}^i \psi_j$ .  $\square$

To get the total cost of all global clusters at the end of the algorithm, we just need to compute  $\Psi_{5+\log_2 \log_2(n/k)}$ , since it was shown in Lemma 6 that the number of rounds is at most  $5 + \log_2 \log_2(n/k)$ .

**Lemma 8.**  $\Psi_{5+\log_2 \log_2(n/k)} = \text{OPT} \cdot O(\ln(n/k))$ .

*Proof.* Note that by Lemma 7,  $\Psi_{5+\log_2 \log_2(n/k)} = \sum_{i=1}^{5+\log_2 \log_2(n/k)} \psi_i$ . By separating the summation into the first term and all others, and noticing that  $n_i$  is decreasing (i.e., all terms with  $n_i \geq 1$  are upper bounded by  $n_1$ ), we get that the first term in the summation is  $3[1 + \ln(n/k)](1 + \epsilon)\text{OPT}$ , and the rest of the terms are  $\text{OPT} \cdot O((\log_2 \log_2(n/k))^2)$ . For large enough  $n/k$ , the first term dominates all of the rest, so for some constant  $\epsilon' > \epsilon$ , the cost is no more than  $3[1 + \ln(n/k)](1 + \epsilon')\text{OPT} = \text{OPT} \cdot O(\log_2(n/k))$ , with small constant terms.  $\square$

Summarizing the above discussion, we have:

**Theorem 2.** There is a polynomial time approximation algorithm for the CMSD $_{\Delta}$  problem that returns at most  $10k$  clusters whose total diameter is at most  $O(\ln(n/k))$  times the optimal solution value with  $k$  clusters.  $\square$

### 3.3 An Approximation Algorithm for CMSD $_{\Delta}$ with Fixed $k$

When  $k$  is fixed, it is possible to obtain a simple 2-approximation algorithm for the CMSD $_{\Delta}$  problem using the transformation shown in Figure 1. We present this result below.

**Theorem 3.** When the number of clusters  $k$  is fixed, there is a 2-approximation algorithm for CMSD $_{\Delta}$ .

*Proof.* The steps of the approximation algorithm are as follows.

1. Using the transformation of Figure 1 construct an instance of the minimum cost set cover problem from the given instance of CMSD $_{\Delta}$  with the parameter  $f$  set to zero.
2. Find a minimum cost set cover consisting of at most  $k$  sets. Since  $k$  is fixed, this step can be done in polynomial time by exhaustive search.
3. If the collection of sets obtained in Step 2 are not pairwise disjoint, then repeatedly merge pairs of sets with nonempty intersection until the collection is pairwise disjoint.
4. Output the collection of sets found in Step 3 as the solution to the CMSD $_{\Delta}$  instance.

Clearly, the approximation algorithm runs in polynomial time. Applying Lemma 2 with  $f = 0$ , the cost of an optimal set cover is at most twice the optimal solution value of the CMSD $_{\Delta}$  instance. Step 2 finds an optimal solution to the set cover problem, and by Lemma 1, the merging operations in Step 3 do not increase the total diameter of the clusters. Thus, the total diameter of the clusters produced is at most twice the optimal value.  $\square$

## 4 Non-approximability Results

### 4.1 Non-approximability without Triangle Inequality

We show that, unless  $P = NP$ , CMSD cannot be efficiently approximated to within any factor even when the number of clusters is fixed at 3. This result can be established through a simple modification to the reduction from Graph 3-colorability (3-COLOR) to CMSD given in [Br78]. We omit this proof due to space constraints.

**Proposition 1.** *Unless  $P = NP$ , for any  $\rho \geq 1$ , no polynomial time algorithm for the CMSD problem can provide a performance guarantee of  $\rho$ .*  $\square$

This non-approximability result should be contrasted with the known result that the CMSD problem is solvable in polynomial time for 2 clusters [HJ87] [MS89].

### 4.2 A Non-approximability Result for $CMSD_{\Delta}$

Here, we prove our non-approximability result for  $CMSD_{\Delta}$ . We establish this result through a reduction from the well known CLIQUE problem [GJ79].

**Proposition 2.** *Unless  $P = NP$ , for any  $\epsilon > 0$ , no polynomial time algorithm for the  $CMSD_{\Delta}$  problem can provide a solution which satisfies the bound on the number of clusters and whose total diameter is within a factor  $2 - \epsilon$  of the optimal value.*

*Proof.* We use a reduction from the CLIQUE problem. Let the undirected graph  $G(V, E)$  and integer  $J \leq |V|$  denote an arbitrary instance of the CLIQUE problem. We construct an instance of the  $CMSD_{\Delta}$  problem consisting of a complete edge weighted graph  $G'$  on the vertex set  $V$  as follows. For any pair of vertices  $u$  and  $v$ , the weight of  $\{u, v\}$  is set to 1 if  $\{u, v\}$  is an edge in  $E$  and to 2 otherwise. Obviously, the resulting edge weights satisfy the triangle inequality. The number of clusters  $k$  is set to  $|V| - J + 1$ . Now, it is straightforward to see that if  $G$  has a clique with  $J$  or more vertices, then  $G'$  can be partitioned into at most  $k$  clusters with a total diameter of 1: the vertices of the clique form one cluster of diameter 1 and each of the remaining  $|V| - J$  vertices forms a separate cluster with a diameter of zero. Further, if  $G$  does not have a clique with  $J$  or more vertices, then any solution with at most  $k$  clusters must have a total diameter of at least 2. The proposition follows.  $\square$

## 5 Other Results

In this section, we briefly mention our other results on the CMSD problem. Details concerning these results will appear in a complete version of the paper.

We have considered the CMSD problem when the underlying graph is a tree with edge weights (rather than a complete graph). In this version, the distance between any pair of nodes is the length of the path between the nodes in the

tree. For this problem, we have developed a polynomial time algorithm using dynamic programming. This algorithm uses  $O(kn^2)$  space and runs in  $O(k^2n^3)$  time. It can also be extended to work for graphs of bounded treewidth.

We have also considered the clustering problem where the goal is to minimize the sum of the radii of the clusters (rather than the sum of the diameters). To discuss these results, we first recall the definition of cluster radius. Let  $C$  be a cluster. For any node  $v$  in  $C$ , let  $d_v$  denote the maximum distance between  $v$  and any other node in  $C$ . The radius of  $C$  is given by  $\min\{d_v : v \in C\}$ . A node  $v$  for which  $d_v$  is equal to the radius of  $C$  is a **center** of  $C$ . When edge weights satisfy the triangle inequality, the diameter of a cluster is at most twice the radius. Therefore, our approximation result for CMSD $_{\Delta}$  (Section 3) carries over (with a different constant within the big-O) to the clustering problem where the goal is to minimize the sum of the radii. We have also been able to show an interesting contrast between the diameter and radius problems for the non-metric case. For fixed  $k$ , while it is NP-hard to obtain even an approximation for the non-metric version of the diameter problem (Section 4.1), the corresponding problem for radius can be solved in polynomial time.

## References

- ABC+98. B. Awerbuch, B. Berger, L. Cowen and D. Peleg, “Near-Linear Time Construction of Sparse Neighborhood Covers”, *SIAM J. Computing*, Vol. 28, No. 1, 1998, pp. 263–277.
- AP98. P. K. Agarwal and C. M. Procopiuc, “Exact and Approximate Algorithms for Clustering”, *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms* (SODA’98), San Francisco, CA, Jan. 1998, pp. 658–667.
- AP00. P. K. Agarwal and C. M. Procopiuc, “Approximation Algorithms for Projective Clustering”, *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms* (SODA’2000), San Francisco, CA, Jan. 2000, pp. 538–547.
- BKK94. V. Batagelj, S. Korenjak-Cerne and S. Klavzar, “Dynamic Programming and Convex Clustering”, *Algorithmica*, Vol. 11, No. 2, Feb. 1994, pp. 93–103.
- Br78. P. Brucker, “On the Complexity of Clustering Problems”, in *Optimization and Operations Research*, Lecture Notes in Economics and Mathematical Systems, Vol. 157, Edited by M. Beckmann and H. Kunzi, Springer-Verlag, Heidelberg, 1978, pp. 45–54.
- CC+97. M. Charikar, C. Chekuri, T. Feder and R. Motwani, “Incremental Clustering and Dynamic Information Retrieval”, *Proc. 29th Annual ACM Symposium on Theory of Computing* (STOC’97), El Paso, TX, May 1997, pp. 626–634.
- CRW91. V. Capoyleas, G. Rote and G. Woeginger, “Geometric Clusterings”, *J. Algorithms*, Vol. 12, No. 2, Jun. 1991, pp. 341–356.
- Da94. A. Datta, “Efficient Parallel Algorithms for Geometric  $k$ -Clustering Problems”, *Proc. 11th Annual Symposium on Theoretical Aspects of Computer Science* (STACS’94), Caen, France, Feb. 1994, Springer-Verlag Lecture Notes in Computer Science, Vol. 775, pp. 475–486.
- DH73. R. Duda and P. Hart, *Pattern Classification and Scene Analysis*, Wiley-Interscience, New York, NY, 1973.

- DKS97. J. S. Deogun, D. Kratsch and G. Steiner, "An Approximation Algorithm for Clustering Graphs with a Dominating Diametral Path", *Information Processing Letters*, Vol. 61, No. 3, Feb. 1997, pp. 121–127.
- FG88. T. Feder and D. H. Greene, "Optimal Algorithms for Approximate Clustering", *Proc. 20th Annual ACM Symposium on Theory of Computing* (STOC'88), Chicago, IL, May 1988, pp. 434–444.
- FPT81. R. Fowler, M. Paterson and S. Tanimoto, "Optimal Packing and Covering in the Plane", *Information Processing Letters*, Vol. 12, 1981, pp. 133–137.
- GH98. N. Guttmann-Beck and R. Hassin, "Approximation Algorithms for Min-sum  $p$ -Clustering", *Discrete Applied Mathematics*, Vol. 89, 1998, pp. 125–142.
- GJ79. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman and Co., San Francisco, CA, 1979.
- Go85. T. F. Gonzalez, "Clustering to Minimize the Maximum Intercluster Distance", *Theoretical Computer Science*, Vol. 38, No. 2-3, Jun. 1985, pp. 293–306.
- HJ87. P. Hansen and B. Jaumard, "Minimum Sum of Diameters Clustering," *Journal of Classification*, Vol. 4, 1987, pp. 215–226.
- HJ97. P. Hansen and B. Jaumard, "Cluster Analysis and Mathematical Programming", *Mathematical Programming*, Vol. 79, Aug. 1997, pp. 191–215.
- Ho97. D. S. Hochbaum (Editor), *Approximation Algorithms for NP-Hard Problems*, PWS Publishing Company, Boston, MA, 1997.
- HS86. D. S. Hochbaum and D. B. Shmoys, "A Unified Approach to Approximation Algorithms for Bottleneck Problems", *J. ACM*, Vol. 33, No. 3, July 1986, pp. 533–550.
- JD88. A. Jain and R. Dubes, *Algorithms for Clustering Data*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.
- KMN99. S. Khuller, A. Moss and J. Naor, "The Budgeted Maximum Coverage Problem", *Information Processing Letters*, Vol. 70, 1999, pp. 39–45.
- Ma99. J. Matousek, "On Approximate Geometric  $k$ -Clustering", Manuscript, Department of Applied Mathematics, Charles University, Prague, Czech Republic, 1999.
- MR+98. M. V. Marathe, R. Ravi, R. Sundaram, S. S. Ravi, D. J. Rosenkrantz and H. B. Hunt III, "Bicriteria Network Design Problems", *J. Algorithms*, Vol. 28, No. 1, July 1998, pp. 142–171.
- MS84. N. Megiddo and K. J. Supowit, "On the complexity of some common geometric location problems," *SIAM J. Computing*, Vol. 13, 1984, pp. 182–196.
- MS89. C. L. Monma and S. Suri, "Partitioning Points and Graphs to Minimize the Maximum or the Sum of Diameters", *Proc. 6th Int. Conf. Theory and Applications of Graphs*, Kalamazoo, Michigan, May 1989.
- Pl82. J. Plesník, "Complexity of Decomposing Graphs into Factors with Given Diameters or Radii", *Math. Slovaca*, Vol. 32, No. 4, 1982, pp. 379–388.
- PRW94. U. Pferschy, R. Rudolf and G. J. Woeginger, "Some Geometric Clustering Problems", *Nordic J. Computing*, Vol. 1, No. 2, Summer 1994, pp. 246–263.
- Ra97. P. Raghavan, "Information Retrieval Algorithms: A Survey", *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms* (SODA'97), Jan. 1997, pp. 11–18.
- ZRL96. T. Zhang, R. Ramakrishnan and M. Livny, "Birch: An Efficient Data Clustering Method for Very Large Databases", *Proc. ACM-SIGMOD International Conference on Management of Data* (SIGMOD'96), Aug. 1996, pp. 103–114.

# Robust Matchings and Maximum Clustering

Refael Hassin and Shlomi Rubinstein

Department of Statistics and Operations Research,  
School of Mathematical Sciences, Tel-Aviv University, Tel-Aviv 69978, Israel.  
`{hassin,shlom}@math.tau.ac.il`

**Abstract.** We consider complete graphs with nonnegative edge weights. A  $p$ -matching is a set of  $p$  disjoint edges. We prove the existence of a maximal (with respect to inclusion) matching  $M$  that contains for any  $p \leq |M|$   $p$  edges whose total weight is at least  $\frac{1}{\sqrt{2}}$  of the maximum weight of a  $p$ -matching. We use this property to approximate graph partitioning problems in which the sizes of the parts of the partitioning are given.

## 1 Introduction

Let  $G = (V, E)$  be a complete graph with vertex set  $V$  such that  $|V| = n$ , edge set  $E$ , and edge weights  $w(u, v) \geq 0$ ,  $(u, v) \in E$ . A  $p$ -matching is a set of  $p$  disjoint edges in a graph. A  $p$ -matching with  $p = \lfloor \frac{n}{2} \rfloor$  is called perfect. A perfect matching  $M$  that contains for any  $p \leq |M|$  a  $p$ -matching whose weight is at least  $\alpha$  times the maximum weight of a  $p$ -matching is said to be  $\alpha$ -robust. We prove that  $G$  contains a  $\frac{1}{\sqrt{2}}$ -robust matching. On the other hand, there are graphs that do not contain an  $\alpha$ -robust matching for any  $\alpha > \frac{1}{\sqrt{2}}$ .

In Section 2 we generalize the robustness concept to independence systems. Our theorem on robust matchings is proved in Section 3 and we use it to approximate within a factor  $\frac{1}{\sqrt{2}}$  the following problem: Given constants  $c_1 \geq c_2 \geq \dots \geq c_p$ , find a  $p$ -matching  $M$  that maximizes  $\sum_{i=1}^p c_i w_i$  where  $w_1 \geq w_2 \geq \dots \geq w_p$  are the edge weights in  $M$ .

In Section 4 we use these results to approximate the MAXIMUM CLUSTERING PROBLEM WITH GIVEN SIZES OF THE SIDES in which the goal is to partition the vertex set to subsets of given sizes maximizing the total edge weight within the same cluster. In the full version of this paper we also apply our results here to approximate within the same bound the MAXIMUM CAPACITATED STAR PACKING PROBLEM in which it is also required to locate a center within each cluster and the goal is to maximize the total distance from each vertex to its center. In both cases we assume that the edge weights satisfy the triangle inequality.

For  $V' \subseteq V$  we denote by  $E(V')$  the edge set of the subgraph induced by  $V'$ . For  $E' \subset E$  we denote by  $W(E')$  the total weight of edges in  $E'$ .

For an optimization problem under consideration we denote by  $opt$  the optimal solution value and by  $apx$  the approximate value returned by a given approximation algorithm. Some of the proofs are omitted in this extended abstract and will appear in the full version of the paper.

## 2 Robust Independent Sets

An *independence system* is a pair  $(E, \mathcal{F})$  consisting of a ground set  $E$  and a collection of *independent sets*, or equivalently, *feasible solutions*, such that  $F' \subset F \in \mathcal{F}$  implies  $F' \in \mathcal{F}$ . Let  $w_e \geq 0$  for  $e \in E$  be weights attached to the elements of  $E$ . The problem of computing an independent set of maximum weight generalizes many interesting combinatorial optimization problems. Korte and Hausmann [4] analyzed the performance of the greedy algorithm for the above problem. The algorithm sorts the elements by weight and inserts them into the solution starting with the heaviest one and excluding an element if its addition would generate a set not in  $\mathcal{F}$ . They proved the following theorem:

**Theorem 2.1** *For any  $E' \subseteq E$  define  $l(E')$  and  $u(E')$  to be the smallest and largest cardinality, respectively, of a maximal (with respect to inclusion) independent set contained in  $E'$ . Let  $r(E, \mathcal{F}) = \min_{E' \subseteq E} \frac{l(E')}{u(E')}$ , then the greedy solution is an  $r(E, \mathcal{F})$ -approximation, that is, the value of the greedy solution is at least  $r(E, \mathcal{F})$  times the optimal value.*

Consider now the following game: You choose a maximal independent set in  $E$ . An adversary then selects  $p \in \{1, \dots, |n|\}$ . Finally, you output the  $p$  heaviest elements of your solution. By the definition of an independence system, the output is independent. Your payoff is the ratio between the weight of your output and the maximum weight of an independent set whose cardinality is at most  $p$ . A solution is  $\alpha$ -robust if it guarantees that the payoff is at least  $\alpha$ .

**Theorem 2.2** *The greedy solution is  $r(E, \mathcal{F})$ -robust.*

The edges and matchings in a graph constitute an independence system for which  $r = \frac{1}{2}$  [4]. It follows that the greedy solution is  $\frac{1}{2}$ -robust. We obtain stronger results in the next section.

Let  $c_1 \geq c_2 \geq \dots \geq c_m \geq 0$  be given constants. For an independent set  $F = \{e_1, \dots, e_m\}$  with weights  $w_1, w_2, \dots, w_m$  define  $C(F) = \sum_{j=1}^m c_j w_j$ . Since we are interested in obtaining large values of  $C(F)$ , we will assume that for any given matching the edges are numbered so that  $w_1 \geq w_2 \geq \dots \geq w_m$ . Thus,  $C(F)$  is well defined for any set  $F$  without explicitly specifying an order on its edges. We will also denote  $F_p = \{e_1, \dots, e_p\}$ ,  $p = 1, \dots, m$  and  $F_p = F$  for  $p > m$ .

*Problem 2.1.* Compute  $F \in \mathcal{F}$ ,  $|F| \leq p$ , that maximizes  $C(F_p)$ .

The following theorem was proved by Gerhard Woeginger [5]:

**Theorem 2.3** *Problem 2.1 is NP-hard even when  $\mathcal{F}$  is the set of matchings in a graph with edge set  $E$  ( $F \subseteq E$  is in  $\mathcal{F}$  if it consists of vertex-disjoint edges).*

**Theorem 2.4** *Let  $F$  and  $F'$  be independent sets. If  $F'$  is  $\alpha$ -robust then  $C(F'_p) \geq \alpha C(F_p)$  for every  $p = 1, 2, \dots$  and any constants  $c_1 \geq c_2 \geq \dots \geq c_m \geq 0$ .*

**Proof:** Let  $w_j = 0$   $j > |F|$ . Let  $w_1 \geq w_2 \geq \dots \geq w_p$  and  $w'_1 \geq w'_2 \geq \dots \geq w'_p$  be the edge weights of  $F_p$  and  $F'_p$ , respectively. Then,

$$\begin{aligned} C(F'_p) &= \sum_{j=1}^{p-1} (c_j - c_{j+1}) \sum_{i=1}^j w'_i + c_p \sum_{i=1}^p w'_i \\ &= \sum_{j=1}^{p-1} (c_j - c_{j+1}) W(F'_j) + c_p W(F'_p) \\ &\geq \sum_{j=1}^{p-1} (c_j - c_{j+1}) \alpha W(F_j) + c_p \alpha W(F_p) \\ &= \alpha \sum_{i=1}^p c_i w_i = \alpha C(F_p). \end{aligned}$$

■

### 3 Robust Matchings

A *matching* is a set of vertex-disjoint edges. The weight of a matching is the total weight of its edges. A *maximum matching* is a matching with maximum weight. A *p-matching* is a matching with  $p$  edges. We denote  $m = \lfloor \frac{n}{2} \rfloor$ , the maximum number of edges in a matching. An  $m$ -matching is said to be *perfect*.

For a perfect matching  $M$  we define  $M_p$  to be the set of its  $p$  heaviest edges,  $p = 1, \dots, m$ . We denote by  $M^{(p)}$  a maximum  $p$ -matching. A matching is  $\alpha$ -robust if

$$W(M_p) \geq \alpha W(M^{(p)}) \quad p = 1, \dots, m.$$

In this section we show that for every graph there exists a  $\frac{1}{\sqrt{2}}$ -robust matching and that it can be constructed by a single application of a maximum matching algorithm. The following example shows that the value of  $\frac{1}{\sqrt{2}}$  cannot be increased.

Consider a 4-vertex graph with weights  $w(1, 2) = w(3, 4) = 1$ ,  $w(2, 3) = \sqrt{2}$  and all other edges have zero weight. For this graph  $W(M_1) = \sqrt{2}$  and  $W(M_2) = 2$ . The graph has three perfect matchings and none is  $\alpha$ -robust for  $\alpha > \frac{1}{\sqrt{2}}$ : The matchings  $\{(1, 2), (3, 4)\}$  and  $\{(2, 3), (1, 4)\}$  are  $\frac{1}{\sqrt{2}}$ -robust and  $\{(1, 3), (2, 4)\}$  is 0-robust.

**Theorem 3.1** *Let  $S$  be a maximum perfect matching with respect to the squared weights  $w^2(e)$   $e \in E$ .  $S$  is  $\frac{1}{\sqrt{2}}$ -robust.*

The rest of this section is devoted to proving Theorem 3.1. We will prove it by treating the squared edge weights as variables whose sizes are to be determined in order to form a contradiction to the theorem. We will prove that to achieve such

a contradiction we may make several assumptions on these variables. Finally these assumptions will lead to the conclusion that the claim is true.

Consider the set  $S \cup M^{(p)}$ . It consists of a collection of disjoint paths and cycles. A path may consist of a single edge or it alternates between  $S$  and  $M^{(p)}$ . Since  $S$  is perfect, the end edges of the path are from  $S$  except possibly one end of one path in the case of odd  $n$  (since in this case there is exactly one vertex which is not incident to an edge of  $S$ .) A cycle alternates between  $S$  and  $M^{(p)}$ . We will construct from the edges of  $S$  a  $p$ -matching whose weight is at least  $\frac{W(M^{(p)})}{\sqrt{2}}$ . Since the weight of this matching is at most the weight of the  $p$  heaviest edges in  $S$ , this construction will prove the theorem.

We choose a  $p$ -matching from  $S$  as follows: Every edge in  $S \cup M^{(p)}$  is chosen. All of the edges of  $S$  contained in a cycle of  $S \cup M^{(p)}$  are chosen. From every nontrivial path (containing more than a single edge) of  $S \cup M^{(p)}$  we choose all the edges that belong to  $S$  except for the lightest one. There is one exception to the last rule: If ( $n$  is odd and) there is a path with only one end edge from  $S$  then we choose all of the  $S$ -edges of this path. The total number of edges selected is equal to  $|M^{(p)}| = p$ . It is sufficient to prove that the claimed bound on the ratio of the edge weights in  $S$  and in  $M^{(p)}$  holds for every such path and cycle.

Consider a nontrivial path  $P$  with squared weights  $x_1, y_1, x_2, y_2, \dots, y_{r-1}, x_r$  where the  $x$  values correspond to the edges of  $S$  and the  $y$  values correspond to the edges of  $M^{(p)}$  in the order they appear on  $P$ .

We denote  $x_{[i,j]} = \sum_{l=i}^j x_l$  and similarly  $y_{[i,j]} = \sum_{l=i}^j y_l$ . We are interested in subpaths  $P_{i,j}$  of  $P$  consisting of the edges whose weights are  $x_i, y_i, \dots, y_{j-1}, x_j$ . Note that  $P = P_{1,r}$ . Since  $S$  is maximum with respect to the squared weights,

$$x_{[i,j]} \geq y_{[i,j-1]} \quad 1 \leq i < j \leq r. \quad (1)$$

Let  $x_{min} = \min\{x_i \mid i = 1, \dots, r\}$ . Our goal is to prove that the ratio of the total weight of the  $r - 1$  heaviest edges in  $P \cap S$  to the weight of  $P \cap M_k$  is at least  $\frac{1}{\sqrt{2}}$ , that is,

$$Z = \frac{\sum_{i=1}^r \sqrt{x_i} - \sqrt{x_{min}}}{\sum_{i=1}^{r-1} \sqrt{y_i}} \geq \frac{1}{\sqrt{2}}$$

for all  $x, y$  that satisfy (1).

We will prove that  $Z \geq \frac{1}{\sqrt{2}}$  for every nontrivial path by induction on  $r$ . Note that the proof and induction hypothesis apply to any nontrivial path  $P$  in  $S \cap M^{(p)}$ , not just to maximal (with respect to inclusion) paths. A subpath is subject to additional constraints arising from longer subpaths that contain it, but these constraints may only increase the lower bound on  $Z$  for the subpath in question.

**Lemma 3.2**  $Z \geq \frac{1}{\sqrt{2}}$  when  $r = 2$ .

**Lemma 3.3**  $Z \geq \frac{1}{\sqrt{2}}$  when  $r = 3$ .

We now proceed to proving the general step of the induction for  $r > 3$ . Thus we assume that the claim holds for smaller  $r$  values.

**Lemma 3.4** *We can assume that  $x_j > x_{\min}$   $j = 2, \dots, r - 1$ .*

**Proof:** Suppose that  $x_j = x_{\min}$  for some  $j \in \{2, \dots, r - 1\}$ . Then,

$$\begin{aligned} Z &= \frac{(\sum_{i=1}^j \sqrt{x_i} - \sqrt{x_{\min}}) + (\sum_{i=j}^r \sqrt{x_i} - \sqrt{x_{\min}})}{\sum_{i=1}^{j-1} \sqrt{y_i} + \sum_{i=j}^{r-1} \sqrt{y_i}} \\ &\geq \min\left\{\frac{\sum_{i=1}^j \sqrt{x_i} - \sqrt{x_{\min}}}{\sum_{i=1}^{j-1} \sqrt{y_i}}, \frac{\sum_{i=j}^r \sqrt{x_i} - \sqrt{x_{\min}}}{\sum_{i=j}^{r-1} \sqrt{y_i}}\right\}. \end{aligned}$$

Since  $x_j = \min\{x_i \mid i = 1, \dots, j\} = \min\{x_i \mid i = j, \dots, r\}$ , it follows from the induction hypothesis that  $Z \geq \frac{1}{\sqrt{2}}$ . ■

We call a subpath  $P_{i,j}$  for which  $x_{[i,j]} = y_{[i,j-1]}$  *tight*.

**Lemma 3.5** (i) *Let  $i \leq k \leq j \leq l$  such that  $i < j$  and  $k < l$ . If  $k < j$  and both  $P_{i,j}$  and  $P_{k,l}$  are tight then so is  $P_{j,k}$ .* (ii) *Let  $i < j < k$ . If  $P_{ij}$  is tight then  $P_{jk}$  isn't.*

**Proof:** (i) By assumption,  $x_{[i,j]} = y_{[i,j-1]}$  and  $x_{[k,l]} = y_{[k,l-1]}$ . Summing these equations we get

$$x_{[i,l]} + x_{[k,j]} = x_{[i,j]} + x_{[k,l]} = y_{[i,j-1]} + y_{[k,l-1]} = y_{[i,l-1]} + y_{[k,j-1]}.$$

Since  $x_{[i,l]} \geq y_{[i,l-1]}$  and  $x_{[k,j]} \geq y_{[k,j-1]}$  it follows that both of the latter relations satisfy equality and the respective subpaths are tight.

(ii) From the same equation with  $j = k$  it follows that  $x_j = 0$  and  $1 < j < r$ , in contrast to Lemma 3.4. ■

Suppose that  $r \geq 3$ . Let  $1 < j < r$ . We can assume that there exists a tight interval containing  $e_j$ , otherwise we reduce  $x_j$  till some subinterval containing  $e_j$  becomes tight, and this change reduces  $Z$ . Consider the intersection of all tight intervals containing  $e_j \in S$ . It follows from Lemma 3.5 that the intersection is a non-trivial tight subpath. Again by these lemma, the  $x$  values in this subpath share the same set of tight subpaths and therefore we can assume that the sum of their squared roots is minimized subject to a single constrain on their sum. By concavity of the square root function, this objective is attained by setting all of these values to 0 except for a single one, say  $x_k > 0$ . From Lemma 3.4 and since  $x_{\min} \geq 0$ , it follows that either  $k \leq 3$  or  $k = 4$ . For the former case the claim has already been proved in Lemmas 3.2 and 3.3. In the latter case, it must be that  $P_{12}$  and  $P_{34}$  are tight and thus  $x_1 = x_4 = y_2 = 0$  while  $y_1 = x_2$  and  $y_3 = x_3$ . In this case  $Z = 1$  and this completes the proof for paths with two ends from  $S$ .

For a path with only one end edge from  $S$  we may assume that a fictitious  $S$ -edge of zero weight is added at that end. The set of constraints (I) then extends in a natural way and the same proof holds.

Suppose now that there is a cycle  $C$  that contradicts the claim. We will show how to construct an instance consisting of a path that contradicts the claim. Since we have already proved that this is impossible, it will follow that such a cycle cannot exist. Specifically, we form a path by cutting  $C$  at an arbitrary vertex and joining many copies of  $C$  pasted at the cut point. Finally add an  $x$  edge at the end where it is missing with a sufficiently large weight, such as  $W(C \cap M^{(p)})$ , so that (II) is satisfied. The path obtained this way will have (asymptotically, as the number of pasted copies increases) the same  $Z$ -value as  $C$ . This concludes the proof of Theorem 3.1.

## 4 Clustering

In the MAXIMUM CLUSTERING PROBLEM, the goal is to partition the vertex set  $V$  into sets of given sizes so that the total weight of edges inside the clusters is maximized. We treat a version of the problem in which cluster sizes  $c_1 \geq c_2 \geq \dots \geq c_p \geq 1$  such that  $c_1 + \dots + c_p = n$  are given. In the *uniform* version,  $c_1 = c_2 = \dots = c_p$ . We consider the problem under the triangle inequality assumption.

Feo and Khellaf [2] treated the uniform case and developed a polynomial algorithm whose error ratio is bounded by  $\frac{c}{2(c-1)}$  or  $\frac{c+1}{2c}$  where  $c = \frac{n}{p}$  is the cluster's size and it is even or odd, respectively. The bound decreases to  $\frac{1}{2}$  as  $c$  approaches  $\infty$ . The algorithm's time complexity is dominated by computation of a maximum weight perfect matching. (Without the triangle inequality assumption, the bound is  $\frac{1}{c-1}$  or  $\frac{1}{c}$ , respectively, but Feo, Goldschmidt and Khellaf [1] improved the bound to  $\frac{1}{2}$  in the cases of  $c = 3$  and  $c = 4$ .) We describe an alternative algorithm for the uniform case that achieves the ratio of  $\frac{1}{2}$  and has a lower  $O(n^2)$  complexity.

Hassin, Rubinstein and Tamir [3] generalized the algorithm of [2] and obtained a bound of  $\frac{1}{2}$  for computing  $k$  clusters of size  $c$  each ( $1 \leq k \leq \frac{n}{c}$ ) with maximum total weight. Our discussion concerning the uniform case does not apply to this generalization.

We first state some results concerning the uniform case. Consider the set of partitions of  $V$  into clusters of size  $c$  each. A random solution is obtained by randomly selecting such a partition.

**Theorem 4.1** *In the uniform case, under the triangle inequality assumption, the expected weight of a random solution is at least  $\frac{1}{2}opt$ .*

The algorithm can be easily derandomized while preserving its performance guarantee.

We now treat the general case. Given  $c_1 \geq c_2 \geq \dots \geq c_p$ , we want to partition  $V$  into clusters of these sizes maximizing their total weight. We note that a random solution may have a very small weight relative to  $opt$ .

Let  $d_j = \lfloor \frac{c_j}{2} \rfloor$ ,  $D_j = d_1 + \dots + d_j$   $j = 1, \dots, p$ , and  $D_0 = 0$ . We propose the following algorithm:

**Algorithm 4.2**

1. Compute a maximum matching  $S$  with respect to the squared weights. Let  $S = \{(u_j, v_j) | j = 1, \dots, m\}$ , where  $w(u_j, v_j) \geq w(u_{j+1}, v_{j+1})$   $j = 1, \dots, m - 1$ .
2. Set  $V_i = \{u_j, v_j | j = D_{i-1} + 1, \dots, D_i\}$   $i = 1, \dots, p$ .
3. For each  $i$  such that  $c_i$  is odd, add to  $V_i$  an arbitrary yet unassigned vertex.

**Theorem 4.3**

$$apx \geq \frac{1}{2\sqrt{2}} opt.$$

**Proof:** Consider an optimal partition  $O_1, \dots, O_p$ . Let  $M_i$  be a maximum matching in the subgraph induced by  $O_i$ ,  $i = 1, \dots, p$ . Denote the edge weights in  $M_i$  by  $w_1^i \geq \dots \geq w_{d_i}^i$ .

Let  $b_i = c_i - 1$  if  $c_i$  is even and  $b_i = c_i$  if  $c_i$  is odd. The edges of  $E(O_i)$  can be covered by a set of  $b_i \leq c_i$  disjoint matchings. Since  $M_i$  is a maximum matching in  $G_i$  it follows that  $b_i W(M_i) \geq W(E(O_i))$  and therefore

$$opt \leq \sum_{i=1}^p c_i W(M_i).$$

Let  $V_1, \dots, V_p$  be the partition produced by Algorithm 4.2. Let  $S_i = S \cap E(V_i)$ . Consider a cluster  $V_i$  with vertices  $u, v, q \in V_i$  such that  $(u, v) \in S_i$ . By the triangle inequality,  $w(u, q) + w(v, q) \geq w(u, v)$ .

Suppose that  $c_i$  is even. Sum this inequality over all  $q \neq u, v \in V_i$ , then sum again over  $(u, v) \in S_i$ . Note that every edge in  $E(V_i) \setminus S_i$  is summed twice. Thus, every edge  $(u, v) \in S_i$  contributes to the total weight of  $E(V_i)$  in addition to its own weight also at least  $\frac{1}{2}(c_i - 2)$  times its weight through the edges incident to it. Thus,  $W(E(V_i)) \geq \frac{1}{2}c_i W(S_i)$ .

Suppose now that  $c_i$  is odd. In this case  $V_i$  contains a vertex, say  $v_i$ , that was added to  $V_i$  in Step 3 of the algorithm. In the summation, the weight of edges incident to  $v_i$  is used just once. Thus, each edge  $(u, v) \in S_i$  contributes its weight  $\frac{1}{2}(c_i - 3)$  times when summed over  $V_i \setminus \{u, v, v_i\}$ , once more through  $w(u, v_i) + w(v, v_i)$ , and once it contributes its own weight. Thus, also in this case,  $W(E(V_i)) \geq \frac{1}{2}c_i W(S_i)$ .

By Theorem 2.4 and the assumption  $c_1 \geq \dots \geq c_p$ ,

$$\begin{aligned} apx &\geq \frac{1}{2} \sum_{i=1}^p c_i W(S_i) \\ &\geq \frac{1}{2\sqrt{2}} \sum_{i=1}^p c_i W(M_i) \\ &\geq \frac{1}{2\sqrt{2}} opt. \end{aligned}$$

■

## References

1. T. Feo, O. Goldschmidt and M. Khellaf, "One half approximation algorithms for the  $k$ -partition problem", *Operations Research* **40**, 1992, S170-S172.
2. T. Feo and M. Khellaf, "A class of bounded approximation algorithms for graph partitioning", *Networks* **20**, 1990, 181-195.
3. R. Hassin, S. Rubinstein and A. Tamir, "Approximation algorithms for maximum dispersion", *Operations Research Letters*, **21** (1997), 133-137.
4. B. Korte and D. Hausmann, "An analysis of the greedy heuristic for independence systems," *Annals of Discrete Mathematics* **2**, 1978, 65-74.
5. G. Woeginger, private communication.

# The Hospitals/Residents Problem with Ties

Robert W. Irving<sup>1</sup>, David F. Manlove<sup>1,\*</sup>, and Sandy Scott<sup>2</sup>

<sup>1</sup> Dept. of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland

{rwi,davidm}@dcs.gla.ac.uk

<sup>2</sup> Dept. of Mathematics, University of Glasgow, Glasgow G12 8QQ, Scotland

ssc@maths.gla.ac.uk

**Abstract.** The hospitals/residents problem is an extensively-studied many-one stable matching problem. Here, we consider the hospitals/residents problem where ties are allowed in the preference lists. In this extended setting, a number of natural definitions for a stable matching arise. We present the first linear-time algorithm for the problem under the strongest of these criteria, so-called *super-stability*. Our new results have applications to large-scale matching schemes, such as the National Resident Matching Program in the US, and similar schemes elsewhere.

## 1 Introduction

The Hospitals/Residents problem (HR) [4,14] is a many-one stable matching problem which is so-named because of its application to large-scale matching schemes, such as the National Resident Matching Program in the US [2], the Canadian Resident Matching Service [1], and the Scottish Pre-registration house officer Allocations (SPA) matching scheme [6]. Each of these centralised schemes administers the annual match of graduating medical students to hospital appointments in its respective country.

An instance of HR involves a set  $\mathcal{R}$  of *residents* and a set  $\mathcal{H}$  of *hospitals*, each resident  $r \in \mathcal{R}$  seeking a post at one hospital, and each hospital  $h \in \mathcal{H}$  having  $q(h) \geq 1$  posts. Each resident in  $\mathcal{R}$  ranks a subset of  $\mathcal{H}$  in strict order, and each hospital  $h \in \mathcal{H}$  ranks its applicants in strict order. An agent  $p \in \mathcal{R} \cup \mathcal{H}$  finds an agent  $q \in \mathcal{R} \cup \mathcal{H}$  *acceptable* if  $q$  appears on  $p$ 's preference list;  $p$  finds  $q$  *unacceptable* otherwise. A *matching*  $M$  is a subset of  $\mathcal{R} \times \mathcal{H}$ , where  $(r, h) \in M$  implies that (i)  $r, h$  find each other acceptable, (ii)  $r$  is assigned to at most one hospital in  $M$ , and (iii) at most  $q(h)$  residents are assigned to  $h$  in  $M$ . A matching  $M$  for an instance of HR is *stable* if  $M$  admits no *blocking pair*. A blocking pair  $(r, h)$  for  $M$  is a resident  $r$  and hospital  $h$  such that (i)  $r, h$  find each other acceptable, (ii)  $r$  either is unassigned or prefers  $h$  to his assigned hospital in  $M$ , and (iii)  $h$  either is undersubscribed or prefers  $r$  to the worst resident assigned to it in  $M$ . If  $(r, h)$  form a blocking pair with respect to a matching  $M$ , then  $(r, h)$  is said to *block*  $M$ . Also, if  $(r, h) \in M$  for some stable matching  $M$ , then we say

\* Supported by Engineering and Physical Sciences Research Council grant number GR/M13329.

that  $(r, h)$  is a *stable pair*, and  $r$  is a *stable partner* of  $h$  (and vice versa). Note that, in view of the definitions of a matching and a blocking pair, we assume throughout this paper, without loss of generality, that an agent  $p$  finds an agent  $q$  acceptable if and only if  $q$  finds  $p$  acceptable. We say that the preference list of a resident  $r \in \mathcal{R}$  (resp. hospital  $h \in \mathcal{H}$ ) is *complete* if  $r$  (resp.  $h$ ) finds all hospitals in  $\mathcal{H}$  (resp. residents in  $\mathcal{R}$ ) acceptable.

The classical Stable Marriage problem (SM) [4][14][8] is a restriction of HR in which each hospital has exactly one post, the number of hospitals equals the number of residents, and all preference lists are complete. For a given instance  $I$  of HR, the Gale/Shapley algorithm for SM [2] may be extended in order to find a stable matching for  $I$  (such a matching in  $I$  always exists) in  $O(mn)$  time, where  $n = |\mathcal{R}|$  and  $m = |\mathcal{H}|$  [4, Section 1.6.3]. The Gale/Shapley algorithm incorporates a sequence of proposals from one set of agents to the other; if the residents propose to the hospitals (the *resident-oriented algorithm*), then we obtain a stable matching  $M$  which is uniquely favourable to the residents: every resident assigned in  $M$  is assigned to his best stable partner, and every resident unassigned in  $M$  is unassigned in any stable matching [4, Section 1.6.3]. Analogously, if the hospitals propose to the residents (the *hospital-oriented algorithm*), then we obtain a stable matching  $M$  which is uniquely favourable to the hospitals: every hospital  $h \in \mathcal{H}$  is assigned either its  $q(h)$  best stable partners, or a set of fewer than  $q(h)$  residents; in the latter case, no other resident is assigned to  $h$  in any stable matching [4, Section 1.6.2].

Although an instance of HR may admit more than one stable matching, every stable matching has the same size, matches exactly the same set of residents, and fills exactly the same number of posts at each hospital; indeed any hospital that is undersubscribed in one stable matching is assigned exactly the same set of residents in all stable matchings. (These results are collectively known as the ‘Rural Hospitals Theorem’ [12][3][13].)

**Ties in the preference lists.** A natural generalisation of HR occurs when each agent’s preference list need not be strictly ordered, but may include ties – we refer to this extension as the Hospitals/Residents problem with Ties (HRT). When ties are permitted, more than one definition of stability is possible [5].

According to the weakest of these stability notions, a matching  $M$  is *weakly stable* [5] if  $M$  admits no blocking pair<sup>1</sup>, where a blocking pair  $(r, h)$  for  $M$  is a resident  $r$  and hospital  $h$  such that (i)  $r, h$  find each other acceptable, (ii)  $r$  either is unassigned or strictly prefers  $h$  to his assigned hospital in  $M$ , and (iii)  $h$  either is undersubscribed or strictly prefers  $r$  to the worst resident assigned to it in  $M$ . Given an instance  $I$  of HRT, the existence of a weakly stable matching is guaranteed: by breaking the ties in  $I$  arbitrarily, we obtain an instance  $I'$  of HR, and clearly a stable matching in  $I'$  is weakly stable in  $I$ . Indeed, a converse of sorts holds, giving the following proposition, whose proof is straightforward and is omitted.

<sup>1</sup> Note that throughout this paper, the form of stability to which the term *blocking pair* refers should be clear from the context.

**Proposition 1.** *Let  $I$  be an instance of HRT, and let  $M$  be a matching in  $I$ . Then  $M$  is weakly stable in  $I$  if and only if  $M$  is stable in some instance  $I'$  of HR obtained from  $I$  by breaking the ties in  $I$  in some way.*

However, the weakly stable matchings in  $I$  may be of different cardinality, and each of the problems of finding the maximum or minimum size of weakly stable matching in an HRT instance is NP-hard, though approximable within a factor of 2 [7,10].

A stronger form of stability may be defined as follows: a matching  $M$  is *super-stable* [5] if  $M$  admits no blocking pair, where a blocking pair  $(r, h)$  for  $M$  is a resident  $r$  and hospital  $h$  such that (i)  $(r, h) \notin M$ , (ii)  $r, h$  find each other acceptable, (iii)  $r$  either is unassigned or strictly prefers  $h$  to his assigned hospital in  $M$  or is indifferent between them, and (iv)  $h$  either is undersubscribed or strictly prefers  $r$  to the worst resident assigned to it in  $M$  or is indifferent between them. Clearly a super-stable matching is weakly stable. Additionally, the super-stability definition gives rise to the following analogue of Proposition 1 (again, the proof is straightforward and is omitted):

**Proposition 2.** *Let  $I$  be an instance of HRT, and let  $M$  be a matching in  $I$ . Then  $M$  is super-stable in  $I$  if and only if  $M$  is stable in every instance  $I'$  of HR obtained from  $I$  by breaking the ties in  $I$  in some way.*

It should be clear that an instance  $I$  of HRT may not admit a super-stable matching: as a simple example, suppose that each hospital has just one post, and every agent's list is a single tie of length 2. It is the purpose of this paper to present optimal  $O(mn)$  algorithms – linear in the size of the problem instance – for determining whether a given instance of HRT admits a super-stable matching, and if it does, to construct such a matching. The first algorithm, presented in Section 2, is resident-oriented in that it involves a sequence of proposals from the residents to the hospitals, and has similar optimality implications for the residents to those of the resident-oriented algorithm for HR. Also in Section 2 we prove an analogue of the Rural Hospitals Theorem for HRT. The second algorithm, presented in Section 3, is the hospital-oriented version, incorporating proposals from the hospitals to the residents, with analogous optimality implications for the hospitals to those of the hospital-oriented algorithm for HR.

For space reasons, the majority of our attention is focused on the resident-oriented algorithm for HRT. It is this algorithm that is likely to be of more significance to implementors of large-scale matching schemes, since recent pressure from student bodies has ensured that all three matching schemes mentioned above essentially employ the resident-oriented algorithm for HR.

**Applications.** Note that permitting ties in the preference lists has important practical applications. In the context of centralised matching schemes, some participating hospitals with many applicants have found the task of producing a strictly ordered preference list difficult, and they have expressed a desire to include ties in their lists. In such a setting, choosing the weak stability definition leads to two problems: (i) finding a weakly stable matching that matches as many

residents as possible, and (ii) the possibility of, say, a resident  $r$  persuading, by some means, a hospital  $h$  to accept  $r$  at the expense of some allocated resident  $r'$ , if  $h$  is indifferent between  $r$  and  $r'$ . The super-stability definition clearly avoids problem (ii), and additionally guards against problem (i), as is demonstrated by the following proposition, which is a consequence of Propositions 1 and 2, and the Rural Hospitals Theorem for HR.

**Proposition 3.** *Let  $I$  be an instance of HRT, and suppose that  $I$  admits a super-stable matching  $M$ . Then the Rural Hospitals Theorem holds for the set of weakly stable matchings in  $I$ .*

Thus Proposition 3 tells us that if a super-stable matching exists, then all weakly stable matchings are of the same size, and match exactly the same set of residents. Of course, as observed earlier, a super-stable matching need not exist. Nonetheless, it is arguable that a super-stable matching should be preferred by a practical matching scheme in cases when one does exist. In Section 4 we address the issue of the existence of super-stable matchings in an HRT instance.

**Previous work.** As mentioned above, optimal algorithms for constructing stable matchings in an instance of HR are known. For the case where ties are permitted, there is an optimal  $O(n^2)$  algorithm, due to Irving [5], for determining whether a given (one-one) instance of Stable Marriage in which preference lists are complete but may incorporate ties (henceforth SMT) admits a super-stable matching, and for constructing one if it does, where  $n$  is the number of men and women. However, the problem of formulating such an algorithm for the (many-one) HRT case has remained open until now.

## 2 Resident-Oriented Algorithm for HRT

For a given instance of HRT, Algorithm HRT-Super-Res, shown in Figure 1, determines whether a super-stable matching exists, and if so will find such a matching. We shall describe informally the execution of Algorithm HRT-Super-Res. Before doing so, we make a number of definitions.

For a given instance  $I$  of HRT, suppose that  $(r, h) \in M$  for some super-stable matching  $M$ . Then  $(r, h)$  is a *super-stable pair*, and  $r$  is a *super-stable partner* of  $h$  (and vice versa). The term *delete the pair*  $(r, h)$ , implies that  $r, h$  are to be deleted from each other's preference lists. By the *head* of a resident's preference list, we mean the set of one or more hospitals, tied in his current list (i.e. his preference list after any deletions have been carried out), which he strictly prefers to all other hospitals in his list. Similarly, the *tail* of a hospital's list refers to the set of one or more residents, tied in its current list, to whom it strictly prefers all other residents in its list. By the term *reduced lists*, we mean the current lists at the termination of Algorithm HRT-Super-Res.

Algorithm HRT-Super-Res involves a sequence of proposals from the residents to the hospitals, in the spirit of the resident-oriented Gale/Shapley algorithm for HR. A resident proposes simultaneously to *all* hospitals at the head

```

assign each resident to be free;
assign each hospital to be totally unsubscribed;
for each hospital  $h$  loop
   $full(h) := \text{false};$ 
end loop;
while some resident  $r$  is free and has a nonempty list loop
  for each hospital  $h$  at the head of  $r$ 's list loop
    provisionally assign  $r$  to  $h$ ; { $r$  "proposes" to  $h$ }
    if  $h$  is oversubscribed then (†)
      for each resident  $s'$  at the tail of  $h$ 's list loop
        if  $s'$  is provisionally assigned to  $h$  then
          break the assignment;
        end if;
        delete the pair  $(s', h)$ ;
      end loop;
    end if;
    if  $h$  is full then (‡)
       $full(h) := \text{true};$ 
       $s :=$  worst resident provisionally assigned to  $h$ ; {any one, if  $> 1$ }
      for each strict successor  $s'$  of  $s$  on  $h$ 's list loop
        delete the pair  $(s', h)$ ;
      end loop;
    end if;
    end loop;
  end loop;
if some resident is multiply assigned or
  (some hospital  $h$  is undersubscribed and  $full(h)$ ) then
    no super-stable matching exists;
  else
    the assignment relation is a super-stable matching;
  end if;

```

**Fig. 1.** Algorithm HRT-Super-Res.

of his list, and all proposals are provisionally accepted. If a hospital  $h$  becomes oversubscribed, it turns out that none of  $h$ 's worst-placed assignees (there must be more than one), nor any residents tied with these assignees in  $h$ 's list, can be a super-stable partner of  $h$  – such pairs  $(r, h)$  are deleted. If a hospital  $h$  is full, then no resident strictly inferior than  $h$ 's worst-placed assignee(s) can be a super-stable partner of  $h$  – again such pairs  $(r, h)$  are deleted. The proposal sequence terminates once every resident either is assigned to a hospital or has an empty list. At this point, it turns out that if a resident is assigned to more than one hospital, or some hospital is undersubscribed but was previously full, then no-super-stable matching exists. Otherwise, the assignment relation is a super-stable matching.

In order to establish the correctness of Algorithm HRT-Super-Res, a number of lemmas follow. The first three of these deal with the case that the assignment

relation is claimed to be a super-stable matching. In what follows,  $I$  is an instance of HRT, in which  $\mathcal{R}$  is the set of residents and  $\mathcal{H}$  is the set of hospitals.

**Lemma 1.** *If, at the termination of the while loop of Algorithm HRT-Super-Res, the algorithm reports that the assignment relation  $M$  is a super-stable matching, then  $M$  is indeed a matching.*

*Proof.* Clearly, no hospital is oversubscribed in  $M$ . Also, no resident is multiply assigned in  $M$ , for otherwise the algorithm would have reported that no super-stable matching exists, a contradiction.  $\square$

**Lemma 2.** *If the pair  $(r, h)$  is deleted during an execution of Algorithm HRT-Super-Res, then that pair cannot block any matching generated by Algorithm HRT-Super-Res, comprising pairs that are never deleted.*

*Proof.* Let  $M$  be a matching generated by Algorithm HRT-Super-Res, comprising pairs that are never deleted, and suppose that  $(r, h)$  is deleted during execution of the algorithm. If  $h$  is full in  $M$ , then  $h$  strictly prefers its worst-placed assignee in  $M$  to  $r$ , since  $r$  is a strict successor of any undeleted entries in the reduced list of  $h$ . Hence  $(r, h)$  does not block  $M$  in this case. Now suppose that  $h$  is undersubscribed in  $M$ . As the pair  $(r, h)$  is deleted by the algorithm, then during some iteration of the while loop,  $h$  must have been full. Hence the algorithm would have reported that no super-stable matching exists rather than generating  $M$ , a contradiction.  $\square$

**Lemma 3.** *If, at the termination of the while loop of Algorithm HRT-Super-Res, the algorithm reports that the assignment relation  $M$  is a super-stable matching, then  $M$  is indeed a super-stable matching.*

*Proof.* By Lemma 1 the assignment relation  $M$  is a matching. Now suppose that  $M$  is blocked by some pair  $(r, h)$ . Then  $r$  and  $h$  are acceptable to each other, so that each is on the original preference list of the other. By Lemma 2, the pair  $(r, h)$  has not been deleted. Hence each is on the reduced list of the other.

As the reduced list of  $r$  is nonempty,  $r$  is assigned to some hospital  $h'$  in  $M$ . Now  $h' \neq h$ , as  $(r, h)$  blocks  $M$ . If  $r$  strictly prefers  $h$  to  $h'$ , then the pair  $(r, h)$  has been deleted, since  $h'$  is at the head of the reduced list of  $r$ , a contradiction. Thus  $r$  is indifferent between  $h$  and  $h'$ , so that  $r$  proposed to  $h$  during the execution of the algorithm. Hence  $r$  is assigned to  $h$  in  $M$ , for otherwise the pair  $(r, h)$  would have been deleted, a contradiction. Thus  $(r, h)$  does not block  $M$ , a contradiction.  $\square$

The next lemma shows that Algorithm HRT-Super-Res will never delete a pair that could belong to some super-stable matching.

**Lemma 4.** *No super-stable pair is ever deleted during an execution of Algorithm HRT-Super-Res.*

*Proof.* Suppose, for a contradiction, that  $(r, h)$  is the first super-stable pair to be deleted during an execution of Algorithm HRT-Super-Res. Let  $M$  be a super-stable matching in  $I$  such that  $(r, h) \in M$ .

*Case (i).* Suppose that  $(r, h)$  is deleted as a result of  $h$  being oversubscribed. Consider the assignment relation  $G$  at point  $(\dagger)$  in the same iteration of the while loop. At this point, some resident  $s$  is provisionally assigned to  $h$  in  $G$ , where  $(s, h) \notin M$  and  $h$  strictly prefers  $s$  to  $r$  or is indifferent between them, since  $(r, h) \in M$  and  $h$  cannot be oversubscribed in  $M$ . There is no super-stable matching in which  $s$  is assigned to a hospital  $h'$  which he strictly prefers to  $h$ . For otherwise, the super-stable pair  $(s, h')$  would have been deleted before  $(r, h)$ , in order for  $s$  to propose to  $h$ , a contradiction. Thus either  $s$  is unassigned in  $M$ , or  $s$  is assigned to  $h'$  in  $M$ , where  $s$  strictly prefers  $h$  to  $h'$  or is indifferent between them. In any of these cases,  $(s, h)$  blocks  $M$ , a contradiction.

*Case (ii).* Suppose that  $(r, h)$  is deleted as a result of  $h$  being full. Consider the assignment relation  $G$  at point  $(\ddagger)$  in the same iteration of the while loop. At this point, some resident  $s$  is provisionally assigned to  $h$  in  $G$ , where  $(s, h) \notin M$  and  $h$  strictly prefers  $s$  to  $r$ , since  $(r, h) \in M$  and  $r$  is not assigned to  $h$  in  $G$ . As in part (i), there is no super-stable matching in which  $s$  is assigned a hospital which he strictly prefers to  $h$ . Thus again,  $(s, h)$  blocks  $M$ , a contradiction.  $\square$

The next two lemmas deal with the case that Algorithm HRT-Super-Res claims the non-existence of a super-stable matching.

**Lemma 5.** *If, at the termination of the while loop of Algorithm HRT-Super-Res, some resident is multiply assigned, then  $I$  admits no super-stable matching.*

*Proof.* Let  $G$  be the assignment relation at the termination of the while loop. Suppose, for a contradiction, that there exists a super-stable matching  $M$  in  $I$ .

Firstly, we claim that some hospital must have fewer assignees in  $M$  than it has provisional assignees in  $G$ . For, suppose not. Let  $p_G(h)$  denote the provisional assignees of hospital  $h$  in  $G$ , and let  $p_M(h)$  denote the assignees of hospital  $h$  in  $M$ , for any  $h \in \mathcal{H}$ . Then by hypothesis,

$$\sum_{h \in \mathcal{H}} |p_M(h)| \geq \sum_{h \in \mathcal{H}} |p_G(h)|. \quad (1)$$

Now if some resident  $r$  is not provisionally assigned to a hospital in  $G$ , then the reduced list of  $r$  is empty, so that by Lemma 4,  $r$  is unassigned in any super-stable matching. Thus, letting  $R_1$  denote the residents who are provisionally assigned to at least one hospital in  $G$ , and letting  $R_2$  denote the residents who are assigned to a hospital in  $M$ , we have  $|R_2| \leq |R_1|$ . Hence

$$\sum_{h \in \mathcal{H}} |p_M(h)| = |R_2| \leq |R_1| < \sum_{h \in \mathcal{H}} |p_G(h)|$$

as some resident is multiply assigned in  $G$ , which contradicts Inequality 1. Thus the claim is established, so that some hospital  $h$  has fewer assignees in  $M$  than

it has provisional assignees in  $G$ . Hence  $h$  is undersubscribed in  $M$ , since no hospital is oversubscribed in  $G$ . In particular, some resident  $r$  is assigned to  $h$  in  $G$  but not in  $M$ . Thus by Lemma 4,  $r$  cannot be assigned to a hospital in  $M$  which he strictly prefers to  $h$ . Hence  $(r, h)$  blocks  $M$ , a contradiction.  $\square$

**Lemma 6.** *If some hospital  $h$  became full during the while loop of Algorithm HRT-Super-Res, and  $h$  subsequently ends up undersubscribed at the termination of the while loop, then  $I$  admits no super-stable matching.*

*Proof.* Let  $G$  be the assignment relation at the termination of the while loop. Suppose, for a contradiction, that there exists a super-stable matching  $M$  in  $I$ . By Lemma 5, no resident is multiply assigned in  $G$ . Let  $h'$  be a hospital which became full during the while loop and subsequently ends up undersubscribed in  $G$ . Then there is some resident  $r'$  who was provisionally assigned to  $h'$  at some point during the while loop, but is not assigned to  $h'$  in  $G$ . Thus the pair  $(r', h')$  was deleted during some iteration of the while loop, so that  $(r', h') \notin M$  by Lemma 4.

Now let  $p_G(h), p_M(h), R_1, R_2$  be defined as in the proof of Lemma 5. Firstly, we claim that if any hospital  $h$  is undersubscribed in  $M$ , then every resident provisionally assigned to  $h$  in  $G$  is also assigned to  $h$  in  $M$ . For, if some resident  $r$  is assigned to  $h$  in  $G$  but not in  $M$ , then  $(r, h)$  blocks  $M$ , since  $h$  is undersubscribed in  $M$ , and by Lemma 4,  $r$  cannot be assigned to a hospital in  $M$  which he strictly prefers to  $h$ .

Secondly, we claim that each hospital has the same number of provisional assignees in  $G$  as it has assignees in  $M$ . For, by the first claim, any hospital that is full in  $G$  is also full in  $M$ , and any hospital that is undersubscribed in  $G$  fills as many places in  $M$  as it does in  $G$ . Hence  $|p_M(h)| \geq |p_G(h)|$  for each  $h \in \mathcal{H}$ . As in the proof of Lemma 5, we also have

$$\sum_{h \in \mathcal{H}} |p_M(h)| = |R_2| \leq |R_1| = \sum_{h \in \mathcal{H}} |p_G(h)|$$

since no resident is multiply assigned in  $G$ . Hence  $|p_M(h)| = |p_G(h)|$  for each  $h \in \mathcal{H}$ .

Thus  $(r', h')$  blocks  $M$ , since  $h'$  is undersubscribed in  $M$  by the second claim, and by Lemma 4,  $r'$  cannot be assigned to a hospital in  $M$  which he strictly prefers to  $h'$ .  $\square$

Together, Lemmas 4–6 establish the correctness of Algorithm HRT-Super-Res. In addition, Lemma 4 implies that there is an optimality property for the partner of a given assigned resident in any super-stable matching output by the algorithm. In particular, we have proved:

**Theorem 1.** *For a given instance of HRT, Algorithm HRT-Super-Res determines whether or not a super-stable matching exists. If such a matching does exist, all possible executions of the algorithm find one in which every assigned resident has as good a partner as in any super-stable matching, and every unassigned resident is unassigned in all super-stable matchings.*

By a suitable choice of data structures, Algorithm HRT-Super-Res can be implemented to run in  $O(mn)$  time and space, where  $m = |\mathcal{H}|$  and  $n = |\mathcal{R}|$ . The time bound follows by noting that the number of iterations of the while loop is bounded by the number of deletions from the preference lists. Note that the complexity of Algorithm HRT-Super-Res can also be expressed in terms of  $L$ , the total length of all preference lists in the HRT instance: clearly the running time is then  $O(L)$ . Since SM is a special case of HRT, the  $\Omega(L)$  lower bound of Ng and Hirschberg [11] for SM implies that Algorithm HRT-Super-Res for HRT is optimal.

We now present the Rural Hospitals Theorem for HRT under super-stability.

**Theorem 2.** *Let  $I$  be a given instance of HRT. Then:*

1. *Each hospital is assigned the same number of residents in all super-stable matchings.*
2. *Exactly the same residents are unassigned in all super-stable matchings.*
3. *Any hospital that is undersubscribed in one super-stable matching is matched with exactly the same set of residents in all super-stable matchings.*

*Proof.* Let  $M, M'$  be two super-stable matchings in  $I$ . Let  $I'$  be an instance of HR obtained from  $I$  by resolving the ties in  $I$  arbitrarily. Then by Proposition 2, each of  $M, M'$  is stable in  $I'$ . By the Rural Hospitals Theorem for stable matchings in an instance of HR [4, Theorem 1.6.3], each hospital is assigned the same number of residents in  $M$  and  $M'$ , exactly the same residents are unassigned in  $M$  and  $M'$ , and any hospital that is undersubscribed in  $M$  is matched with exactly the same set of residents in  $M'$ .  $\square$

### 3 Hospital-Oriented Algorithm for HRT

In this section, we consider the hospital-oriented analogue of Algorithm HRT-Super-Res, namely Algorithm HRT-Super-Hosp, shown in Figure 2. We begin by describing the execution of Algorithm HRT-Super-Hosp informally.

Algorithm HRT-Super-Hosp involves a sequence of proposals from the hospitals to the residents, in the spirit of the hospital-oriented Gale/Shapley algorithm for HR. A hospital  $h$  proposes simultaneously to the most preferred resident  $r$  on  $h$ 's list not already provisionally assigned to  $h$ , and to all other residents tied with  $r$  in  $h$ 's list. These proposals are provisionally accepted. If a resident  $r$  becomes multiply assigned and is indifferent between his provisional assignees, it turns out that neither of  $r$ 's provisional assignees, nor any hospitals tied with them in  $r$ 's list, can be a super-stable partner of  $r$  – such pairs  $(r, h')$  are deleted. If a resident  $r$  receives a proposal from a hospital  $h$ , then no hospital  $h'$  to whom  $r$  strictly prefers  $h$  can be a super-stable partner of  $r$  – again such pairs  $(r, h')$  are deleted. The proposal sequence terminates once every hospital is either full or provisionally assigned to everyone on its current list. At this point, it turns out that if a hospital is oversubscribed, or some resident is unassigned but was previously provisionally assigned, then no super-stable matching exists. Otherwise, the assignment relation is a super-stable matching.

```

assign each resident to be free;
assign each hospital to be totally unsubscribed;
for each resident  $r$  loop
   $assigned(r) := \text{false};$ 
end loop;
while some hospital  $h$  is undersubscribed and
   $h$ 's list contains a resident  $r'$  not provisionally assigned to  $h$  loop
     $r' :=$  most preferred such resident in  $h$ 's list; {any one, if  $> 1$ }
    for each resident  $r$  tied with  $r'$  in  $h$ 's list loop {including  $r'$ }
      provisionally assign  $r$  to  $h$ ; { $h$  "proposes" to  $r$ }
       $assigned(r) := \text{true};$ 
      if  $r$  is multiply assigned and
         $r$  is indifferent between his provisional assignees then
          for each hospital  $h'$  at the tail of  $r$ 's list loop
            if  $r$  is provisionally assigned to  $h'$  then
              break the assignment;
            end if;
            delete the pair  $(r, h');$ 
          end loop;
        else
          for each strict successor  $h'$  of  $h$  on  $r$ 's list loop
            if  $r$  is provisionally assigned to  $h'$  then
              break the assignment;
            end if;
            delete the pair  $(r, h');$ 
          end loop;
        end if;
      end loop;
    end if;
  end loop;
end loop;
if (some resident  $r$  is not assigned and  $assigned(r)$ ) or
some hospital is oversubscribed then
  no super-stable matching exists;
else
  the assignment relation is a super-stable matching;
end if;

```

**Fig. 2.** Algorithm HRT-Super-Hosp.

In order to establish the correctness of Algorithm HRT-Super-Hosp, a number of lemmas follow. We omit the proofs, which use similar techniques to those of Section 2. We begin by stating the analogues of Lemmas 3 and 4 for Algorithm HRT-Super-Hosp. In what follows,  $I$  is an instance of HRT, in which  $\mathcal{R}$  is the set of residents and  $\mathcal{H}$  is the set of hospitals.

**Lemma 7.** *If, at the termination of the while loop of Algorithm HRT-Super-Hosp, the algorithm reports that the assignment relation  $M$  is a super-stable matching, then  $M$  is indeed a super-stable matching.*

**Lemma 8.** *No super-stable pair is ever deleted during an execution of Algorithm HRT-Super-Hosp.*

The next two lemmas deal with the case that Algorithm HRT-Super-Hosp claims the non-existence of a super-stable matching.

**Lemma 9.** *If, at the termination of the while loop of Algorithm HRT-Super-Hosp, some hospital is oversubscribed, then  $I$  admits no super-stable matching.*

**Lemma 10.** *If some resident  $r$  became assigned during the while loop of Algorithm HRT-Super-Hosp, and  $r$  subsequently ends up unassigned at the termination of the while loop, then  $I$  admits no super-stable matching.*

Together, Lemmas 7–10 establish the correctness of Algorithm HRT-Super-Hosp. In addition, Lemma 8 implies that there is an optimality property for the assignees of a given fully-subscribed hospital in any super-stable matching output by the algorithm. In particular, we have proved:

**Theorem 3.** *For a given instance of HRT, Algorithm HRT-Super-Hosp determines whether or not a super-stable matching exists. If such a matching does exist, all possible executions of the algorithm find one in which every hospital  $h \in \mathcal{H}$  is assigned either its  $q(h)$  best super-stable partners, or a set of fewer than  $q(h)$  residents; in the latter case, no other resident is assigned to  $h$  in any super-stable matching.*

As is the case for Algorithm HRT-Super-Res, by considering suitable data structures, Algorithm HRT-Super-Hosp can be implemented to run in  $O(mn)$  time and space, where  $m = |\mathcal{H}|$  and  $n = |\mathcal{R}|$ . Again, the time bound follows by noting that the number of iterations of the while loop is bounded by the number of deletions from the preference lists. Note that the complexity of Algorithm HRT-Super-Hosp can also be expressed in terms of  $L$ , the total length of all preference lists in the HRT instance: clearly the running time is then  $O(L)$ . As is the case for Algorithm HRT-Super-Res, this time bound is optimal.

## 4 Existence of Super-stable Matchings

Algorithm HRT-Super-Res has been implemented and some preliminary experiments have been carried out, in order to give an indication of the likelihood of a super-stable matching existing in a given HRT instance. There are clearly several parameters that can be varied in these tests, such as the numbers of residents and hospitals, the capacities of the hospitals, the lengths of the preference lists, and the number, position and sizes of the ties. A range of vectors of values for the aforementioned parameters were considered, and for each vector, a set of random instances was created, each satisfying the particular constraints on the instance. Finally, the percentage of instances in each set admitting a super-stable matching was computed.

Perhaps not surprisingly, the empirical results suggest that the probability of a super-stable matching existing decreases as the size of the instance increases, and also decreases as the number and length of the ties increase. However, it was found that the probability of a super-stable matching existing is likely to be much higher if the ties occur on one side only, for example in the hospitals' lists and not in the residents' lists (further details may be found in [15]). This is a situation that is likely to occur naturally in practice: for example, in the context of resident/hospital matching schemes, residents are typically asked to rank a relatively small number of hospitals, and might find it easier to produce a strictly ordered preference list than would a large hospital with many applicants.

Due to the large number of different parameters that can be varied in empirical tests, clearly such experiments cannot hope to provide a comprehensive analysis of the likelihood of a super-stable matching existing in an arbitrary HRT instance. It remains open to establish theoretical bounds on the probability of a super-stable matching existing in a given random instance of HRT.

## 5 Concluding Remarks

In this paper we have highlighted the importance of the super-stability criterion in HRT, with reference to large-scale matching schemes. Current practice in the SPA scheme, for example, is that hospitals are permitted to express ties in their preference lists. However, any ties are broken arbitrarily so as to give an instance with strictly ordered lists. Hence by Proposition 1 the SPA scheme will produce matchings that can only guarantee to be weakly stable in the original instance. We suggest that such centralised matching schemes should first search for a super-stable matching using Algorithm HRT-Super-Res, and only if none exists should they settle for a weakly stable matching.

We finish with an open problem. A third stability criterion, so-called *strong stability*, can be applied to an HRT instance [5]. In the strong stability case, the definition of a blocking pair is similar to that of the super-stability case, except that at most one agent in the pair is permitted to express indifference between the other agent and its (possibly worst) partner(s) in the matching. Clearly a super-stable matching is strongly stable, and a strongly stable matching is weakly stable. Additionally, the strong stability and super-stability definitions coincide if the ties belong to the preference lists of one set of agents only. As is the case for super-stability, a given instance of HRT may not admit a strongly stable matching (see [5] for further details). However, there is an  $O(n^4)$  algorithm, due to Irving [5], for determining whether a given instance of SMT admits a strongly stable matching, and for constructing one if it does, where  $n$  is the number of men and women. An extended version of this algorithm, also of  $O(n^4)$  complexity, has been formulated by Manlove for SMTI (the variant of SMT in which preference lists may be incomplete) [9]. We leave open the problem of constructing a polynomial-time algorithm, or establishing NP-completeness, for HRT under strong stability.

## References

1. Canadian Resident Matching Service. How the matching algorithm works. Web document available at <http://www.carms.ca/algorith.htm>.
2. D. Gale and L.S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69:9–15, 1962.
3. D. Gale and M. Sotomayor. Some remarks on the stable matching problem. *Discrete Applied Mathematics*, 11:223–232, 1985.
4. D. Gusfield and R.W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, 1989.
5. R.W. Irving. Stable marriage and indifference. *Discrete Applied Mathematics*, 48:261–272, 1994.
6. R.W. Irving. Matching medical students to pairs of hospitals: a new variation on an old theme. In *Proceedings of ESA '98: the Sixth European Symposium on Algorithms*, volume 1461 of *Lecture Notes in Computer Science*, pages 381–392. Springer-Verlag, 1998.
7. K. Iwama, D. Manlove, S. Miyazaki, and Y. Morita. Stable marriage with incomplete lists and ties. In *Proceedings of ICALP '99: the 26th International Colloquium on Automata, Languages, and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 443–452. Springer-Verlag, 1999.
8. D.E. Knuth. *Stable Marriage and its Relation to Other Combinatorial Problems*, volume 10 of *CRM Proceedings and Lecture Notes*. American Mathematical Society, 1997. English translation of *Mariages Stables*, Les Presses de L’Université de Montréal, 1976.
9. D.F. Manlove. Stable marriage with ties and unacceptable partners. Technical Report TR-1999-29, University of Glasgow, Department of Computing Science, January 1999.
10. D.F. Manlove, R.W. Irving, K. Iwama, S. Miyazaki, and Y. Morita. Hard variants of stable marriage. Technical Report TR-1999-43, University of Glasgow, Department of Computing Science, September 1999. Submitted for publication.
11. C. Ng and D.S. Hirschberg. Lower bounds for the stable marriage problem and its variants. *SIAM Journal on Computing*, 19:71–77, 1990.
12. A.E. Roth. The evolution of the labor market for medical interns and residents: a case study in game theory. *Journal of Political Economy*, 92(6):991–1016, 1984.
13. A.E. Roth. On the allocation of residents to rural hospitals: a general property of two-sided matching markets. *Econometrica*, 54:425–427, 1986.
14. A.E. Roth and M.A.O. Sotomayor. *Two-sided matching: a study in game-theoretic modeling and analysis*, volume 18 of *Econometric Society Monographs*. Cambridge University Press, 1990.
15. S. Scott. Implementation of matching algorithms. Master’s thesis, University of Glasgow, Department of Computing Science, 1999.

# Incremental Maintenance of the 5-Edge-Connectivity Classes of a Graph

Yefim Dinitz<sup>1</sup> and Ronit Nossenson<sup>2</sup>

<sup>1</sup> Dept. of Computer Science  
Ben-Gurion University, Beer-Sheva, 84105, Israel  
[dinitz@cs.bgu.ac.il](mailto:dinitz@cs.bgu.ac.il)

<sup>2</sup> Dept. of Computer Science  
Technion, Haifa, 32000, Israel  
[ronitt@cs.technion.ac.il](mailto:ronitt@cs.technion.ac.il)

**Abstract.** Two vertices of an undirected graph are called  $k$ -edge-connected if there exist  $k$  edge-disjoint paths between them. The equivalence classes of this relation are called  $k$ -edge-connected classes, or  $k$ -classes for short. This paper shows how to check whether two vertices belong to the same 5-class of an arbitrary connected graph that is undergoing edge insertions. For this purpose we suggest (i) a full description of the 4-cuts of an arbitrary graph and (ii) a representation of the  $k$ -classes,  $1 \leq k \leq 5$ , of size linear in  $n$ —the number of vertices of the graph; these representations can be constructed in a polynomial time. Using them, we suggest an algorithm for incremental maintenance of the 5-classes. The total time for a sequence of  $m$  *Edge-Insert* updates and  $q$  *Same-5-Class?* queries is  $O(q + m + n \cdot \log^2 n)$ ; the worst-case time per query is  $O(1)$ .

## 1 Introduction

Connectivity is a fundamental property of graphs which is used in network reliability analysis, in network design problems, and other applications. In 1990's, dynamic maintenance and augmentation of high connectivity has become an important area of research (see, e.g., [12, 7, 8, 9, 12, 15, 16, 20]). One of the directions concerns 1-, 2-, ...,  $k$ -connectivity in an arbitrary graph. As for motivation, designers of communication networks are usually interested now in analysis and maintenance for small values of  $k$ , even 1 or 2, since networks of higher connectivity are too expensive. Theorists, on their side, try to extend their techniques as far as possible to be ahead of today needs, as usual. However, complexity of graph structures grows tremendously with  $k$  growing. In this paper we show that the case  $k = 5$  still admits a compact description and a more or less efficient incremental algorithm. However, as far as we see, this is due to some lucky combination of existing methods, while the approach is not like to be extendible to greater connectivities.

Let  $G = (V, E)$  be an undirected connected multi-graph without loops. A minimal edge-cut  $C$  of  $G$  (cut, for short) is an edge set whose removal disconnects  $G$  and removal of any proper part of  $C$  does not disconnect  $G$ . If  $|C| = k$  then

$C$  is called a  $k$ -cut. Two vertices  $\{u, v\}$  are called  $k$ -edge-connected if no  $k'$ -cut,  $k' < k$ , separates  $u$  from  $v$ . It is well known that the property “there exist  $k$  edge-disjoint paths between  $u$  and  $v$  in  $G$ ” defines the same relation (see [13]). The equivalence classes of this relation are called the  $k$ -edge-connected classes ( $k$ -classes, for short). The partition of  $V$  into the  $(k+1)$ -classes is a refinement of the partition of  $V$  into  $k$ -classes. Thus, the connectivity classes have an hierarchical structure.

In this paper we are concerned with the problem of maintaining the  $k$ -classes of  $G$  under edge insertions, i.e., *incremental maintenance*. The main tool for solving such a problem is a certain abstract model that describes the graph connectivity structure in a way that enables to decide efficiently how this structure changes when the graph is modified. Such a model must represent not only the connectivity classes but also the system of all minimal cuts that form these classes (for example, the Gomory-Hu tree [11], which presents a bounded subsystem of such cuts, cannot serve for efficient incremental maintenance).

Efficient algorithms for the problem of incremental maintenance of the 1-, 2-, 3- and 4-classes are known [20, 12, 15, 9]. Westbrook and Tarjan [20] used the bridge-tree of a graph to handle its 2-classes. Galil and Italiano [12] and, independently, La Poutre, Leeuwen and Overmars [15] used the “cycle-tree” model of a 2-connected graph to describe and maintain the 3-classes of a connected graph. The well known cactus tree model [6] represents the  $(\lambda + 1)$ -classes of an arbitrary  $\lambda$ -connected graph and is used for their incremental maintenance [9] (the bridge-tree and the cycle-tree are, in fact, special cases of this model.) In [7], the 2-level cactus tree model was introduced. It generalizes the cactus tree model to represent the  $(\lambda + 1)$ - and  $(\lambda + 2)$ -classes of any  $\lambda$ -connected graph and serves for their incremental maintenance.

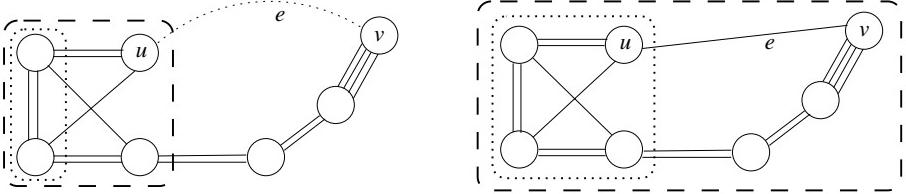
Paper [9] uses the cactus tree model to maintain the 1-, 2-, 3- and 4-classes. The main innovation used is a special kind of a graph object: the 3-component  $\bar{A}$  corresponding to a 3-class  $A$  [4].<sup>1</sup> The graph  $\bar{A}$  has  $A$  as the vertex set and mimics the connectivity structure of  $A$  in a localized fashion: it contains all edges of  $G$  between vertices in  $A$ , and also an edge between each pair of vertices in  $A$  that are connected by a path that travels entirely through vertices outside  $A$ . The 3-component is 3-connected and its cactus tree model is a tree; this tree represents all 4-classes of  $G$  contained in  $A$ . In [9], the problem of incremental maintenance of the 1-, 2-, 3-, and 4-classes is hierarchically decomposed into subproblems on  $k'$ -components of  $G$ ,  $1 \leq k' \leq 3$ .

In general, there are several difficulties in the incremental maintenance of the connectivity classes (see Figure 1 for illustration). The insertion of an edge  $e = (u, v)$  into  $G$  can affect the structure associated with a  $k'$ -class that contains neither  $u$  nor  $v$ . Furthermore, the changes in such a structure are not necessarily as simple as those caused by the insertion of an edge between two vertices in  $A$ .

---

<sup>1</sup> In the literature classes of  $k$ -edge-connectivity are sometimes called  $k$ -edge-connected components. Following the common tradition concerning 1- and 2-components and [18, 14, 4, 9] concerning 3-components, we use the term “component” for a *graph* related to such a class.

However, paper [9] shows that the changes for each individual  $k'$ -class,  $1 \leq k' \leq 4$ , can be performed with only minimal knowledge of the connectivity structures of the rest of  $G$ . When several  $k$ -classes merge, the model for the joint  $k$ -class is be constructed, based on models of the constituting classes.



**Fig. 1.** Changes in the connectivity classes resulting from an edge insertion. (A dashed line encircles one 3-class and a dotted line one 4-class, in each graph.)

In fact, the new light of [9] and [7] is crucial for analysis of dynamics of 5-connectivity. Our work is the construction of entire building on the cornerstones worked out in these papers. Our main contribution made during this construction is as follows.

Concerning statics, we suggest, for the first time, a full description of the 4-cuts of an arbitrary graph, thus filling the gap between descriptions of [7] and [9].

Generalizing the approach of [9], we associate the 2-level cactus tree model, instead of the cactus model, with each 3-component  $\bar{A}$  of  $G$ , to describe the 4- and 5-classes contained in  $A$ . We extend the localized transformation of cactus tree models suggested in [9] to 2-level cactus tree models, based on the incremental maintenance algorithm [7]. Dynamics are much more complicated in our case than in [9], since the 2-level cactus tree model is substantially more complicated than the cactus tree model.

The (amortized) complexities of the incremental algorithms for 1, 2 and 3-classes depend on the number of updates as the  $\alpha$ -function, while the algorithm [9] has an  $O(n \log n)$  term. Our algorithm has an  $O(n \log^2 n)$  term instead of it. To achieve this bound, we suggest a new technique that effectively handles a wide spectrum of dynamic forest operations including both arbitrary tree linking and breaking-into-two the cycle order of the children of a tree vertex (for the first time, to the best of our knowledge). The sizes of all above models are linear in  $|V|$ .

This paper is organized as follows. Section 2 brings basic definitions and notations. Section 3 presents the static description of our model. Section 4 deals with the model dynamics; in particular, in Section 4.2, a general example is given. Section 5 describes the main ideas of the implementation.

## 2 Preliminaries and Notations

Let  $G = (V, E)$  be an undirected connected multi-graph without loops, where  $|V| = n \geq 2$  and  $|E| = m$ . In this paper we refer only to edge-cuts, so the prefix “edge-” is omitted from now on. We refer to “minimal cuts” as simply “cuts”, unless otherwise is mentioned. 1-cuts are referred as **bridges**. The family of all  $k$ -cuts of  $G$  is denoted by  $F^k$ .

Let  $X, Y \subset V$ , the set of edges with one end-vertex in  $X$  and the other in  $Y$  is denoted by  $\delta(X, Y)$ ; obviously,  $\delta(X, Y) = \delta(Y, X)$ . We also denote  $V \setminus X$  by  $\bar{X}$ . Any cut  $C$  corresponds to the unique 2-partition  $(X, \bar{X})$  of  $V$  such that  $C = \delta(X, \bar{X})$  [7]. So we can refer to a cut by the 2-partition defining it.

We say that a cut  $C = \delta(X, \bar{X})$  **divides**  $S$ ,  $S \subseteq V$  (or that  $C$  is an  $S$ -cut), if both  $X \cap S$  and  $\bar{X} \cap S$  are nonempty. We say that a cut divides a subgraph if it divides its vertex set. A subset  $S$  of  $V$ ,  $|S| \geq 2$ , is  **$k$ -connected** if there exist in  $G$   $k$  edge-disjoint paths between every two vertices in  $S$ , that is, there are no  $S$ -cuts of cardinality less than  $k$  in  $G$ . The **connectivity**  $\lambda(S)$  of  $S$  is defined to be the maximum  $k$  for which  $S$  is  $k$ -connected, or in other words, the connectivity of  $S$  is the minimum number of edges in an  $S$ -cut in  $G$ . The connectivity of  $G$  is defined to be  $\lambda(V)$ , denoted for short by  $\lambda$ .

For any  $S \subseteq V$ , the **induced subgraph**  $G(S)$  consists of the vertices in  $S$  and edges in  $E$  connecting vertices in  $S$ . To **shrink** a subset of vertices  $S \subseteq V$  means to replace  $G(S)$  by a single new vertex  $s$ , and, for every edge with one end-vertex in  $S$ , to replace this end-vertex by  $s$ ; any edge of a new graph is identified with its corresponding edge of  $G$ . Let  $C = (v_1, v_2, \dots, v_r, v_1)$ ,  $r \geq 2$ , be a cycle. To **squeeze** a cycle at  $v_i$  and  $v_j$ ,  $i < j$ , is to shrink the set  $\{v_i, v_j\}$  to a new node  $v$ . This operation creates two new cycles from the old cycle  $C$ :  $(v, v_{j+1}, \dots, v_r, v_1, \dots, v)$  and  $(v, v_{i+1}, \dots, v_{j-1}, v)$ . Each of the new cycles may degenerate to the vertex  $v$ . To **contract** an edge  $e = (u, v)$  means to shrink the set  $\{u, v\}$ . To **break** an edge  $e = (u, v)$  by a vertex  $x$  means to add a new vertex  $x$  to  $G$ , and to replace the edge  $e$  by two new edges  $(u, x)$  and  $(x, v)$ .

For a given partition  $\mathcal{P}$  of  $V$ , the related **quotient graph** is defined to be the result of shrinking each part of  $\mathcal{P}$  into a single vertex. The **quotient mapping**  $f_{\mathcal{P}}$ , defined on  $V$ , takes any vertex in such a part  $W$  to the vertex given by shrinking  $W$ . We denote by  $Q^k(G)$  the quotient graph of  $G$  generated by shrinking each  $k$ -class, and by  $f^k$  the corresponding quotient mapping. It is easy to show that for any  $1 \leq k' \leq k-1$ ,  $f^{k'}$  provides a bijection between the  $k'$ -cuts of  $Q^k(G)$  and the  $k'$ -cuts of  $G$ . The **bridge-tree** of a connected graph  $G$  is  $Q^2(G)$ , see [15][20]. A **bridge-path** is an ordered sequence of bridges of a graph which forms a path in its bridge-tree. Similarly, the **cycle-tree** of a 2-connected graph is  $Q^3(G)$  (see [12][15]). A **cycle-path** in a cycle-tree is an ordered sequence of its cycles such that any two consequent cycles have (a single) common vertex.

For a family  $F$  of cuts of  $G$ , the equivalence classes of the relation “ $\{x, y\}$  is not divided by any cut in  $F$ ” are called  **$F$ -atoms**. A **cut model** for  $G$  and a family  $F$  of cuts of  $G$ , is a triple  $(\mathcal{G}, \psi, \mathcal{F})$  as follows. The connected graph  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$  is called the **structure graph**. We refer to its vertices as **nodes**. The mapping  $\psi : V \rightarrow \mathcal{V}$  is called the **structure mapping**. For every

node  $\mathcal{N}$  of  $\mathcal{G}$ ,  $\psi^{-1}(\mathcal{N})$  is either an  $F$ -atom or the empty set. If  $\psi^{-1}(\mathcal{N}) = \emptyset$  we say that  $\mathcal{N}$  is an empty node. We say that a cut  $\mathcal{C} = \delta(\mathcal{X}, \bar{\mathcal{X}})$  of  $\mathcal{G}$   $\psi$ -induces a cut  $\psi^{-1}(\mathcal{C}) = \delta(\psi^{-1}(\mathcal{X}), \psi^{-1}(\bar{\mathcal{X}}))$  of  $G$  if both  $\psi^{-1}(\mathcal{X})$  and  $\psi^{-1}(\bar{\mathcal{X}})$  are nonempty.  $\mathcal{F}$  is a family of cuts of  $\mathcal{G}$ , called the **modeling family**, such that  $\psi^{-1}(\mathcal{F}) = F$ . Observe that, for any cut model, shrinking a subset of nodes of  $\mathcal{G}$  implies naturally a new model: its mapping is the composition of the original mapping and the quotient one.

The bridge-tree mentioned above is a simple example of a cut model. The set of its 1-cuts represents bijectively the set of 1-cuts of  $G$ . Another example of a cut model, for a 2-connected graph, is the cycle-tree mentioned above. The set of its 2-cuts represents bijectively the set of 2-cuts of  $G$ .

The **cactus tree model** [6] is a cut model for the family of minimum cuts of a graph and the  $(\lambda + 1)$ -classes which  $V$  is "cut into" by these cuts. The cactus tree model is defined by the triple  $(\mathcal{H}, \varphi, \mathcal{F})$  as follows. The structure graph  $\mathcal{H} = (\mathcal{V}_{\mathcal{H}}, \mathcal{E}_{\mathcal{H}})$  is a tree-of-edges-and-cycles graph. It is a connected graph s.t. every edge belongs to at most one cycle (in other words, every block of  $\mathcal{H}$  is an edge or a cycle); such a graph is called a **cactus tree**. In the case  $\lambda$  is odd,  $\mathcal{H}$  is cycle-free, meaning, it is a tree. For every node  $\mathcal{N}$  of  $\mathcal{H}$ ,  $\varphi^{-1}(\mathcal{N})$  is either a  $(\lambda + 1)$ -class of  $G$  or the empty set. The modeling family  $\mathcal{F}$  is the family of minimal cuts of  $\mathcal{H}$ , and  $\varphi^{-1}(\mathcal{F}) = F^{\lambda}$ . The number of edges in  $\mathcal{H}$  is linear in the number of  $(\lambda + 1)$ -classes of  $G$ , i.e., is  $O(n)$ . For an algorithm of construction of the cactus tree model see [10] (its complexity is  $O(m + \lambda^2 n \cdot \log(m/n))$ ).

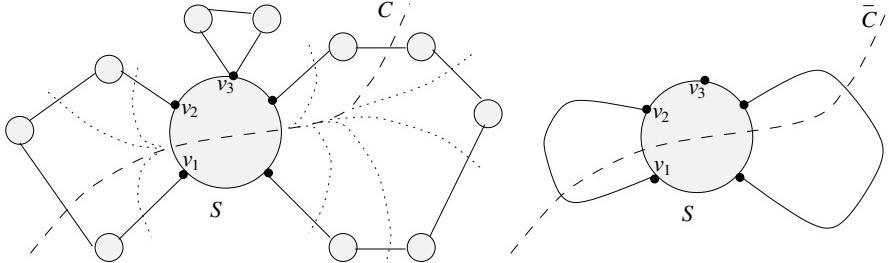
### 3 Model Description

In order to construct a model for representation and incremental maintenance of the  $k$ -classes of a graph,  $1 \leq k \leq 5$ , we combine two known models. The first one, suggested by Dinitz and Westbrook in [9], represents the  $k$ -classes of a graph,  $1 \leq k \leq 4$ , and serves to maintain this representation under edge insertions. The second one, the 2-level cactus tree mode of Dinitz and Nutov [7], serves the same purpose for  $4 \leq k \leq 5$ , for a 3-connected graph.

In this section we follow [4, 9, 7]. The definition of the structure is done via decomposition of the graph into auxiliary graphs, called components. The model has an hierarchic structure which uses the above mentioned models: the bridge-tree, the cycle-tree, the cactus tree and the 2-level cactus-tree. Consider a connected graph  $G$ . Its associated bridge-tree provides a complete description of the 2-classes in  $G$ . The **2-component** associated with a 2-class,  $A$ , is the induced graph  $\bar{A} = G(A)$ . The cycle-tree model of each 2-component is used to describe its 3-classes. By [12, 15], the collection of all cycle-tree models provides a complete description of the 3-classes in  $G$ .

Let  $S$  be a 3-class contained in a 2-class  $H$ . Consider the cycle-tree of  $H$ ,  $Q^3(\bar{H})$ . Let  $L$  be a cycle of  $Q^3(\bar{H})$  incident to  $S$  and let  $(u, v_1)$  and  $(w, v_2)$ ,  $v_1, v_2 \in S$ , be the edges of  $G$  incident to  $S$  on  $L$  (see Figure 2). The vertices  $v_1$  and  $v_2$  are called the **attachment** vertices. In the case they are distinct, the **virtual edge**  $e_S(L)$  is defined as  $(v_1, v_2)$ . The **3-component**  $\bar{S}$  associated with

$S$  is defined as the induced graph  $G(S)$  together with the virtual edges defined by all such cycles. The 3-component  $\bar{S}$  mimics the connectivity structure of  $S$  in a localized fashion in the following sense.



**Fig. 2.** The 3-component.

**Theorem 1** ([4]). *The 3-component  $\bar{S}$  has the following properties:*

- (i) *It is 3-connected;*
- (ii) *For each  $k$ -cut  $C$  of  $G$  dividing  $S$ ,  $k \geq 3$ , there is a  $k'$ -cut  $\bar{C}$ ,  $k' \leq k$ , of  $\bar{S}$  dividing  $S$  in the same way;*
- (iii) *For each  $k$ -cut  $\bar{C}$  of  $\bar{S}$ ,  $k \geq 3$ , there is a nonempty set ("bunch") of  $k$ -cuts of  $G$  dividing  $S$  in the same way as  $\bar{C}$  does.*

It is easy to deduce from this theorem that the  $k$ -classes of  $G$ ,  $k \geq 3$ , contained in  $S$  are exactly the  $k$ -classes of  $\bar{S}$ . Hence, all we need for the description is an appropriate model for 4- and 5-classes of a 3-connected graph. Indeed, then the collection of such models for all 3-components provides a full description of 4- and 5-classes of  $G$ . Paper [9] uses cactus tree models of 3-components to describe and maintain the 4-classes. For the 5-classes we use, instead of it, its extension: the 2-level cactus tree model [7] (the case  $\lambda = 3$ ). Such a model, for a graph, is as follows.

**Theorem 2** ([7]). *In the case  $\lambda = 3$ , for  $F^3 \cup F^4$  there exists a cut model  $(\mathcal{H}^2, \varphi^2, \mathcal{F}^2)$  of size  $O(n)$ , with the following properties:*

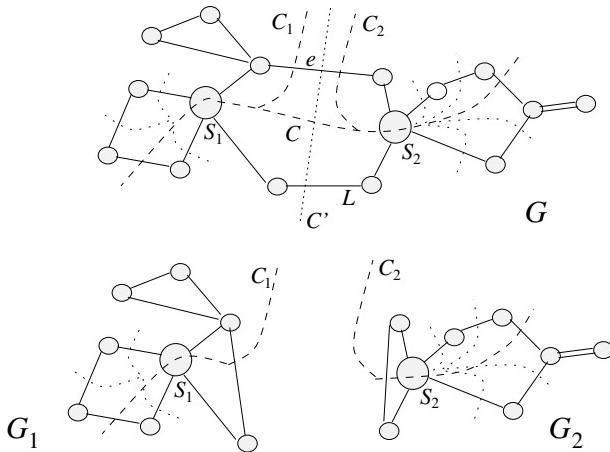
- (i) *The structural graph  $\mathcal{H}^2$  is a tree of edges, cycles, and cube graphs, such that each node of each cube graph is empty and is incident to exactly one bridge;*
- (ii) *The modeling family  $\mathcal{F}^2$  consists of:*
  - *the 1-cuts (bridges);*
  - *the minimal 2-cuts which are all pairs of edges of any block of  $\mathcal{H}^2$  that is a cycle.*
  - *for any block of  $\mathcal{H}^2$  that is a cube graph, the three cuts consisting each of four its pairwise nonadjacent edges;*

- the non-minimal 2-cuts of  $\mathcal{H}^2$  of the form  $\{\{\varepsilon', \varepsilon''\} : \varepsilon', \varepsilon'' \in \hat{P}, \hat{P} \in \Pi\}$ , where  $\Pi$  is a certain set of bridge-paths of  $\mathcal{H}^2$ , such that any two of them have at most one edge in common.
- (iii) The mapping  $(\varphi^2)^{-1}$  takes the set of 1-cuts bijectively onto  $F^3$  and the set of other cuts in  $\mathcal{F}^2$  onto  $F^4$ .

Let us get more information from paper [7]. If we shrink each 2-class of the 2-level cactus tree  $\mathcal{H}^2$  into a single node, we obtain a model which is isomorphic to the cactus tree model of  $G$ . It is a tree since  $\lambda = 3$  is odd; let us denote it by  $\mathcal{H}$ . For  $e = (u, v) \in E$ ,  $\hat{P}(e)$  denotes the bridge-path between  $\varphi(u)$  and  $\varphi(v)$  in  $\mathcal{H}$ . The set of the bridge-paths  $\Pi$  consists of the paths  $\hat{P}(e)$  for all edges  $e$  such that  $|\hat{P}(e)|$  contains at least two edges.

Paper [7] provides an efficient algorithm for incremental maintenance of 5-classes of a 3-connected graph. However, the case of a general graph is more complicated: in particular, an edge insertion causes, in general, merging of 3-classes; thus, we need an algorithm for merging 2-level cactus models.

For the general case, we suggest the following full description of 4-cuts of an arbitrary graph  $G$  (see Figure 3 for illustration).



**Fig. 3.** Example for 4-cuts description.

**Theorem 3 ([5]).** For an arbitrary graph,  $G$ , the set of its 4-cuts consists of the following two sub-families:

- i) The 4-cuts dividing a single 3-class. This sub-family is decomposed into bunches corresponding to the 4-cuts  $\bar{C}$  of  $\bar{S}$ , for all 3-components  $S$ . Any such bunch is the result of all possible independent substitution in  $\bar{C}$  of every virtual edge  $e_S(L)$  by any edge from the cycle  $L$ .

- (ii) The 4-cuts dividing exactly two 3-classes. Such classes are 3-classes  $S_1$  and  $S_2$  incident to the same cycle  $L$  of  $Q^3(G)$ , each by two distinct attachment vertices, such that there exist 3-cuts  $\bar{C}_1$  and  $\bar{C}_2$  of the 3-components  $\bar{S}_1$  and  $\bar{S}_2$ , respectively, containing the virtual edges  $e_{S_1}(L)$  and  $e_{S_2}(L)$ , respectively. This sub-family is decomposed into bunches corresponding to the above pairs of 3-cuts. Any such bunch is the result of all possible independent substitution in  $\bar{C}_1 \cup \bar{C}_2 \setminus \{e_{S_1}(L), e_{S_2}(L)\}$  of every virtual edge  $e_S(L')$ ,  $L \neq L'$  by any edge from the cycle  $L'$ .

**Sketch of proof.** The proof for the case (i) is based on Theorem 11.

Assume a 4-cut  $C$  divides some two 3-classes  $S_1$  and  $S_2$ . Then there exists a 2-cut  $C'$  separating  $S_1$  from  $S_2$ . Let us “divide”  $G$  into two auxiliary graphs  $G_1$ ,  $G_2$  as in Figure 3. Then  $C$ , together with an edge  $e \in C'$ , generates two cuts: an  $r$ -cut  $C_1$  of  $G$  and of  $G_1$  and a  $q$ -cut  $C_2$  of  $G$  and of  $G_2$ , where  $C_1 \cup C_2 = C$ ,  $C_1 \cap C_2 = \{e\}$ ; hence,  $r + q = 6$ . Since both cuts divide 3-classes, holds  $r, q \geq 3$ ; hence,  $r = q = 3$ . By 4, any 3-cut of  $G$  containing an edge  $e$  in a 2-cut (i) divides a single 3-class and (ii) this 3-class belongs to the (unique) cycle of the cycle-tree model that  $e$  belongs to. Therefore,  $C$  divides exactly two 3-classes  $S_1$  and  $S_2$ , these 3-classes belong to the same cycle,  $L$ , of the cycle-tree, and cuts  $C_1$  and  $C_2$  contain the edges  $e_{S_1}(L)$  and  $e_{S_2}(L)$ , respectively. Figure 3 presents an example, with the bunch of 4-cuts defined by  $C$ . (Comment: notice that cuts  $C_1$  and  $C_2$ , and thus also  $C$ , do not divide any 4-class.)

## 4 Model Dynamics

Let us now turn to the model dynamics under insertion of an edge into  $G$ . Meaning, we support updates  $Insert-Edge(u, v)$  where  $u, v \in V$ . Note that as a result of an edge insertion, the only possible kind of modification in the connectivity classes is that a subset of existing  $k$ -classes merges into a single  $k$ -class. For proofs of the statements given below see [19].

**Theorem 4.** *Let  $S \subseteq V$  be a  $k$ -class. The insertion of an edge between two vertices in  $S$  can increase connectivity between vertices which belong to  $S$  only.*

Following 9,7, we perform separately changes in each connectivity level. In each level we translate the changes to several “local” changes. The changes are done on single components, not on the entire  $G$ , and then the results in each connectivity level are combined.

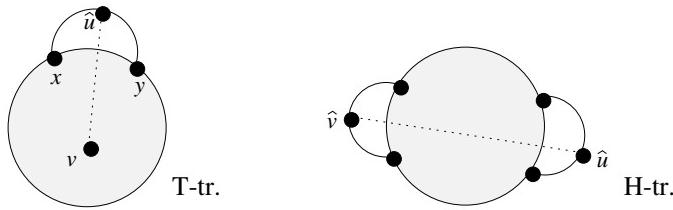
Paper 9 provides the incremental maintenance of the  $k$ -classes,  $1 \leq k \leq 4$ , of a connected graph. Paper 7 provides the incremental maintenance of the 5-classes of a 3-connected graph. Let us see what extension is needed for the incremental maintenance of the 5-classes of a general connected graph. We distinguish the following three cases of insertion of a new edge: the vertices  $u$  and  $v$  are 3-, 2- or only 1-connected.

In the first case, let  $u, v$  belong to some 3-class  $S$ . By theorem 4, no model changes, except for the 2-level cactus-tree model of  $\bar{S}$ . The algorithm of transformation of a 2-level cactus-tree model is given in 7.

In the third case, all three connectivity levels are influenced. By [9], the entire transformation is reduced to a certain transformation of the bridge-tree and to separate transformations for certain “involved” 2-components. Each one of the latter transformations is exactly the same as if a certain edge is inserted into the 2-component [9]. Thus we have a reduction to the second case.

The second case includes most of our work. In this case we insert a new edge  $e$  between two vertices  $u$  and  $v$  in the same 2-class  $H$  of  $G$  but  $v \in S$ ,  $u \in T$ , where  $S, T$  are distinct 3-classes contained in  $H$ . By Theorem [4], this insertion does not increase connectivity between vertices which do not belong to  $H$ . In the cycle-tree  $Q^3(G(H))$ , each node represents a 3-class. The **involved** 3-classes are  $S, T$  and all 3-classes that separate between them in the cycle-tree. In this case we change two levels of connectivity. At the first level, we correct the 2-component  $\bar{H}$  and its model according to [9]. At the second level we correct the involved 3-components and their models (by [9, Lemma 16], non-involved 3-components do not change resulting such an edge insertion.)

Paper [9] uses, as intermediate objects, results of certain **T-transformations** of the 3-components  $\bar{S}$  and  $\bar{T}$  and their related models, and certain **H-transformations** of the other involved 3-components and their related models (for illustration see Figure 4). The T-transformation is caused by breaking a certain edge  $e_u$  by a new vertex  $\hat{u}$  and adding the new edge  $(v, \hat{u})$ . The H-transformation is caused by breaking certain edges  $e_u$  and  $e_v$  by new vertices  $\hat{u}$  and  $\hat{v}$ , respectively, and adding the new edge  $(\hat{v}, \hat{u})$ .



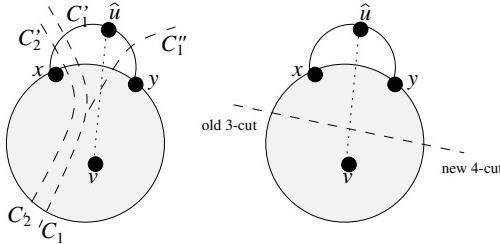
**Fig. 4.** The T- and H-transformations.

Paper [9] provides procedures for updating the cactus tree model of the involved 3-components in T- and H-transformations. We generalize these transformations to a 2-level cactus tree model. Paper [9] also provides a procedure for merging the involved 3-components and their cactus tree models. We generalize this procedure to merge 2-level cactus-tree models. Our procedures are much more complicated than in [9], since the 2-level cactus tree model is substantially more complicated than the cactus tree model (see Theorem [2]).

In this extended abstract we describe only T-transformation. The proof of its correctness, as well as descriptions and proofs for H-transformation and for the merge procedure can be found in [19].

#### 4.1 T-transformation

Let us see what changes occur in the 3,4-cuts of  $\bar{S}$  as the result of a T-transformation, where  $e_u = (x, y)$  (see Figure 5). Consider 3-cuts. First, a new 3-cut that separates  $\hat{u}$  from the rest of  $\bar{S}$  is added. Every 3-cut  $C_1 = (A, B)$  such that  $x \in A$  and  $y, v \in B$  creates two cuts. The first is the 3-cut  $C'_1 = (A, B \cup \{\hat{u}\})$  which contains edge  $(x, \hat{u})$  instead of edge  $e_u$ . The second is 4-cut  $C''_1 = (A \cup \{\hat{u}\}, B)$  which contains edges  $(\hat{u}, y)$  and  $(v, \hat{u})$  instead of edge  $e_u$ . Each 3-cut which separates  $v$  from both  $x$  and  $y$  gets the new edge  $(v, \hat{u})$  and becomes a 4-cut. Consider 4-cuts. Every 4-cut  $C_2 = (A, B)$  such that  $x \in A$  and  $y, v \in B$  creates 4-cut  $C'_2 = (A, B \cup \{\hat{u}\})$  which contains the edge  $(x, \hat{u})$  instead of the edge  $e_u$ . Every 4-cut which separates  $v$  from both  $x$  and  $y$  gets the new edge  $(v, \hat{u})$ , becomes a 5-cut and goes out of consideration.

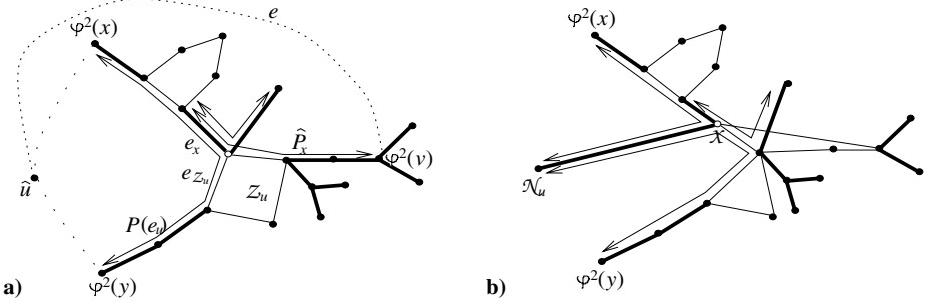


**Fig. 5.** Behavior of 3,4-cuts under a T-transformation.

Let us describe the transformed 2-level cactus tree model. In this paper we consider only the case where no block of the model is a cube graph (the general case is considered in [19]). We use the following notation (for illustration see Figure 6). The path-of-edges-and-cycles between  $\varphi^2(x)$  and  $\varphi^2(y)$  in  $\mathcal{H}^2$  is denoted by  $P(e_u)$ . Its bridges form, in general, a path which belongs to  $\Pi$ , denoted by  $\hat{P}(e_u)$ . The shortest path of bridges and cycle-edges between  $\varphi^2(x)$  and  $\varphi^2(y)$  is denoted by  $\tilde{P}(e_u)$ . We denote the node or cycle which belongs to  $P(e_u)$  and is the nearest to  $\varphi^2(v)$  by  $\mathcal{Z}_u$ . We define  $P_{uv}$  to be the path of edges-and-cycles between  $\mathcal{Z}_u$  and  $\varphi^2(v)$ . The bridge which is on the path between  $\varphi^2(x)$  (resp.,  $\varphi^2(y)$ ) and  $\mathcal{Z}_u$  and is the nearest to  $\mathcal{Z}_u$  (if exists) is denoted by  $e_x$  (resp.,  $e_y$ ). Note that  $e_x, e_y \in P(e_u)$ . By [7, Fact 5.1], in the case  $\mathcal{Z}_u$  is a cycle,  $\mathcal{Z}_u$  has exactly one common cycle-edge with  $\tilde{P}(e_u)$ ; we denote this cycle-edge by  $e_{\mathcal{Z}_u}$ .

Consider a path  $\hat{P}$  of  $\Pi$  which intersects both  $P(e_u)$  and  $P_{uv}$ . The sub-path  $(\hat{P} \cap (P(e_u) \cup P_{uv}))$  is called the **intersection cycle-generating sub-path (ICGS-path, for short)** defined by  $\hat{P}$ . In general, we have four cases.  $\mathcal{Z}_u$  can be a node or a cycle. In each case we have two possibilities: there exist or do not exist ICGS-paths. Let us learn more about the ICGS-paths.

**Lemma 1.** *There exist at most two distinct ICGS-paths, and if there exist two, then at least one of them has exactly two bridges.*



**Fig. 6.** Example of T-transformation in the case  $\mathcal{Z}_u$  is a cycle and there exists an ICGS-path: (a) the original model; (b) the transformed model. (Two-arrowed lines mark paths in  $\Pi$ .)

We use the following notation. By definition, each ICGS-path includes exactly one of  $\{e_x, e_y\}$ . We assume, w.l.o.g., that if there exists a single ICGS-path, it contains  $e_x$ ; we denote it by  $\hat{P}_x$ . In the case there are two ICGS-paths, we denote by  $\hat{P}_y$  someone which has exactly two bridges. By Lemma 1, we have six cases:  $\mathcal{Z}_u$  is a node and there exist either zero or one or two ICGS-paths, or  $\mathcal{Z}_u$  is a cycle and there exist either zero or one or two ICGS-paths. The following statements show that the third option can be reduced to the second one (so it is not taken care of below), and that the sixth option cannot take place.

**Lemma 2.** *If  $\mathcal{Z}_u$  is a node and there exist two distinct ICGS-paths, then the 4-cut of  $S$  which is  $\varphi^2$ -induced by the single non-minimal 2-cut defined by  $\hat{P}_y$  is represented twice in the model.*

**Lemma 3.** *If  $\mathcal{Z}_u$  is a cycle then there exists at most one ICGS-path.*

Following is the description of the  $T$ -transformation of the 2-level cactus tree model of  $\bar{S}$  (for illustration see Figure 6).

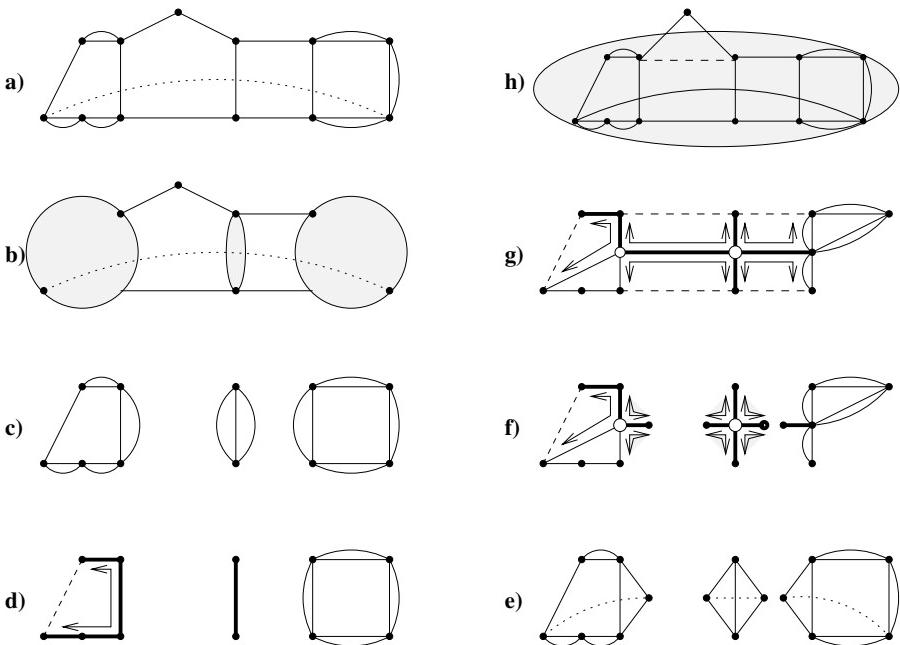
#### 2-level-T-transformation( $\bar{S}, e_u, \varphi^2(v)$ ):

1. Find  $P(e_u)$ ,  $\mathcal{Z}_u$ ,  $e_{\mathcal{Z}_u}$ ,  $P_{uv}$ ,  $e_x$ , and  $\hat{P}_x$ ;
2. If  $\hat{P}_x$  exists then break  $e_x$  by a new empty node  $\mathcal{X}$ ;
3. Else if  $\mathcal{Z}_u$  is a cycle then break  $e_{\mathcal{Z}_u}$  by a new empty node  $\mathcal{X}$ ;  
Else ( $\hat{P}_x$  does not exist and  $\mathcal{Z}_u$  is not a cycle) denote  $\mathcal{Z}_u$  by  $\mathcal{X}$ ;
4. Perform Algorithm 7( $\bar{S}, \mathcal{X}, \varphi^2(v)$ );
5. Add a new node denoted by  $\mathcal{N}_u$ , and a new bridge  $(\mathcal{N}_u, \mathcal{X})$ ;
6. Replace path  $\hat{P}(e_u)$  in  $\Pi$  by the two following paths:
  - The part of  $\hat{P}(e_u)$  between  $\varphi^2(x)$  and  $\mathcal{X}$  (if not empty) plus  $(\mathcal{N}_u, \mathcal{X})$ ;
  - The part of  $\hat{P}(e_u)$  between  $\varphi^2(y)$  and  $\mathcal{X}$  (if not empty) plus  $(\mathcal{N}_u, \mathcal{X})$ ;
7. Return( $\mathcal{N}_u, \mathcal{X}$ );

The update of the structural mapping is as follows. For any vertex  $w \in V$  such that  $\varphi^2(w) = \mathcal{N}_w$  and the node  $\mathcal{N}_w$  has been shrunk (with some other nodes) into a node  $\mathcal{N}'_w$ , the new image of  $w$  is  $\mathcal{N}'_w$ . The image of the new node  $\hat{u}$  is  $\mathcal{N}_u$ .

## 4.2 General Example

In Figure 7, a general example is given. Part (a) presents the original graph  $G$ , with the new edge shown by a dotted line. In part (b), there is given the 3-quotient of  $G$ , which has 4 nodes; the three nodes-nontrivial 3-classes are shown by shadowed circles and ellipse. Part (c) contains the three involved 3-components; the four virtual edges are the four vertical arcs in the middle of the figure.



**Fig. 7.** General example.

The 2-level cactus tree models for these 3-components are presented in part (d). The left and the middle models are, basically, the cactus tree models. In addition, there is one path in  $\Pi$  in the left model, shown by the two-arrowed line; the corresponding edge of  $G$  is shown dashed. There and further, thicker structural edges belong to the cactus tree model of  $G$ , while thinner edges form cycles representing 4-cuts (there are four such cycles of length 2 each in the right model).

Parts (e-h) go upwards in the Figure, so that all the parts are placed in a cyclic order. In part (e), the updated involved 3-components are shown; the updates in the left and right models cause T-transformations, while the update of the middle model causes an H-transformation. Part (f) presents the 2-level cactus tree models of the updated 3-components; the left and right models result from the algorithm given in Section 4.1.

In part (g), these models are merged into the 2-level cactus tree model of the united 3-component. This 3-component corresponds to the single nontrivial 3-class of the incremented graph  $G$ ; this 3-class is the union of the three involved 3-classes of  $G$ . Part (h) shows the two-node 3-quotient of the incremented graph. Inside the shadowed node, the new 3-component is shown; its single virtual edge—corresponding to the pair of edges going to the other (trivial) 3-component—is shown dashed.

One more comment on indication of paths in  $\Pi$ . Two-arrowed lines are longer than necessary for pointing to all bridges in such a paths, they go along cycle edges of the model too. This is done to show—by the ends of such a line—to where are mapped the two end-vertices of the edge defining the path in  $\Pi$ , by the structural mapping.

## 5 Implementation

For implementation of the above transformations we extend the technique of [9], which uses dynamic trees of [17]. The main difficulty is that model graphs associated with 3-components are not trees, as in [9], but cactus trees. As a result, the `squeeze-cycle` operation must be added. We implement as follows. First of all, a cycle  $L$  in a cactus tree is represented, as in [12, 9], by a special dummy vertex with an edge from it to every vertex of  $L$ , plus the list of vertices of  $L$  in the cycle order. In this way a cactus tree is implemented as a tree.

Suppose we have to squeeze a cycle  $L$  at its vertices  $x$  and  $y$ , resulting in two cycles  $L_1$  and  $L_2$ . Assume  $|L_1| \leq |L_2|$ . We scan  $L$  from  $x$  in both directions step by step. In time  $O(|L_1|)$  we arrive at  $y$ , thus finding  $L_1$ . Using  $O(|L_1|)$  dynamic tree operations, we pull all vertices of  $L_1$  out of  $L$  forming two separate cycles  $L_1$  and  $L_2$  with a single common vertex, instead of  $L$ , as required.

In order to bound time, we amortize the above dynamic tree operations on vertex histories. Since  $|L_1|$  is at most  $|L|/2$  above, any vertex changes the cycle it belongs to at most  $\log n$  times. Since a dynamic tree operation costs  $O(\log n)$ , the total time sums to  $O(n \log^2 n)$ .

**Theorem 5.** *The 5-classes of an arbitrary graph can be maintained under any sequence of  $q$  `Same-5-Class( $u, v$ )?` queries and  $m$  `Insert-Edge( $u, v$ )` updates in total time  $O(q + m + n \cdot \log^2 n)$ . The worst-case time per query is  $O(1)$ .*

## References

1. A. A. Benczur, “Augmenting undirected connectivity in  $\tilde{O}(n^3)$  time”, *Proc. 26th Annual ACM Symp. on Theory of Computing*, ACM Press, 1994, 658–667.
2. A. A. Benczur, “The structure of near-minimum edge cuts”, In *Proc. 36st Annual Symp. on Foundations of Computer Science*, 1995, 92–102.
3. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, McGraw-Hill, New York, NY, 1990.

4. Ye. Dinitz, “The 3-edge components and the structural description of all 3-edge cuts in a graph”, *Proc. 18th International Workshop on Graph-Theoretic Concepts in Computer Science (WG92), Lecture Notes in Computer Science*, v.657, Springer-Verlag, 1993, 145–157.
5. Ye. Dinitz, “The 3-edge components and the structural description of edge cuts in a graph”, Manuscript.
6. E. A. Dinic, A. V. Karzanov and M. V. Lomonosov, “On the structure of the system of minimum edge cuts in a graph”, *Studies in Discrete Optimization*, A. A. Fridman (Ed.), Nauka, Moscow, 1976, 290–306 (in Russian).
7. Ye. Dinitz and Z. Nutov, “A 2-level cactus tree model for the minimum and minimum+1 edge cuts in a graph and its incremental maintenance”, *Proc. the 27th Symposium on Theory of Computing*, 1995, 509–518.
8. Ye. Dinitz and A. Vainshtein, “The connectivity carcass of a vertex subset in a graph and its incremental maintenance”, *Proc. 26th Annual ACM Symp. on Theory of Computing*, ACM Press, 1994, 716–725 (see also TR-CS0804 and TR-CS0921, Technion, Haifa, Israel).
9. Ye. Dinitz and J. Westbrook, “Maintaining the Classes of 4-Edge-Connectivity in a Graph On-Line”, *Algorithmica* **20** (1998), no. 3, 242–276.
10. H. N. Gabow, “Applications of a poset representation to edge connectivity and graph rigidity”, *Proc. 23rd Annual ACM Symp. on Theory of Computing*, 1991, 112–122.
11. R. E. Gomory and C. T. Hu, “Multi-terminal network flows”, *J. SIAM* **9**(4) (1961), 551–570.
12. Z. Galil and G. F. Italiano, “Maintaining the 3-edge-connected components of a graph on line”, *SIAM J. Computing* **22** (1), 1993, 11–28.
13. F. Harary. *Graph Theory*, Addison-Wesley, Reading, MA, 1972.
14. Hopcroft, J., and Tarjan, R.E., Dividing a graph into triconnected components. *SIAM J. Comput.* **2** (1973) 135–158.
15. J. A. La Poutré, J. van Leeuwen, and M. H. Overmars. Maintenance of 2-and 3-edge-connected components of graphs. *Discrete Mathematics* **114**, 1993, 329–359.
16. D. Naor, D. Guisfield and C. Martel, “A fast algorithm for optimally increasing the edge connectivity”, In *Proc. 31st Annual Symp. on Foundations of Computer Science*, 1990, 698–707.
17. D. D. Sleator and R. E. Tarjan, “A data structure for the dynamic trees”, In *Proc. 13th Annual ACM Symposium on Theory of Computing*, 1981, 114–122.
18. Tutte, W.T., *Connectivity in Graphs*, Univ. of Toronto Press, Toronto, 1966.
19. Teplixke, R.<sup>2</sup> *Dynamic Maintenance of Connectivity Classes of a Graph, Using Decomposition into 3-Components*, M. Sc. Thesis, the Technion, Haifa, Israel, 1999.
20. J. Westbrook and R. E. Tarjan, “Maintaining bridge-connected and biconnected components on line”, *Algorithmica*, **7** (1992), 433–464.

<sup>2</sup> Teplixke is the maiden name of Ronit Nossenson.

# On the Minimum Augmentation of an $\ell$ -Connected Graph to a $k$ -Connected Graph

Toshimasa Ishii and Hiroshi Nagamochi

Department of Information and Computer Science,  
Toyohashi University of Technology,  
Aichi 441-8580, Japan.  
`{ishii,naga}@ics.tut.ac.jp`

**Abstract.** Given an undirected graph  $G = (V, E)$  and a positive integer  $k$ , we consider the problem of augmenting  $G$  by the smallest number of new edges to obtain a  $k$ -vertex-connected graph. In this paper, we show that, for  $k \geq 4$  and  $k \geq \ell + 2$ , an  $\ell$ -vertex-connected graph  $G$  can be made  $k$ -vertex-connected by adding at most  $\delta(k-1) + \max\{0, (\delta-1)(\ell-3)-1\}$  surplus edges over the optimum in  $O(\delta(k^2n^2 + k^3n^{3/2}))$  time, where  $\delta = k - \ell$  and  $n = |V|$ .

## 1 Introduction

The problem of augmenting a graph by adding the smallest number of new edges to meet vertex-connectivity requirements has been extensively studied as an important subject in the network design problem [4], the data security problem, [13], the graph drawing problem [12] and others, and many efficient algorithms have been developed so far.

Given an undirected graph  $G = (V, E)$  and a positive integer  $k$ , we consider the problem of augmenting  $G$  by the smallest number of new edges to obtain a  $k$ -vertex-connected ( $k$ -connected, for short) graph. We call this problem *the  $k$ -vertex-connectivity augmentation problem* ( $k$ -VCAP, for short). Currently it is known that  $k$ -VCAP for  $k \in \{2, 3, 4\}$  can be solved in polynomial time ([27], [617] and [5] for  $k = 2, 3$  and 4, respectively), where an initial graph  $G$  may not be  $(k-1)$ -connected. For an arbitrary integer  $k > 0$ , whether  $k$ -VCAP is polynomially solvable or not is still an open question (even if an initial graph is restricted to be  $(k-1)$ -connected). When an initial graph is  $(k-1)$ -connected, Jordán presented an  $O(n^5)$  time approximation algorithm for  $k$ -VCAP with a general  $k$  [10, 11]. The difference between the number of new edges added by his algorithm and the optimal value is at most  $(k-2)/2$ .

However, it was an open question whether there exists a good approximation algorithm for  $k$ -VCAP if an initial graph is not  $(k-1)$ -connected. For arbitrary integers  $k$  and  $\delta \geq 2$ , we consider whether there exists a polynomial time algorithm that makes a given  $(k-\delta)$ -connected graph  $k$ -connected by adding a set  $E'$  of new edges such that the difference between  $|E'|$  and the optimal value  $opt$  is small, say  $|E'| - opt = O(\delta k)$ . One may apply Jordán's algorithm  $\delta$  times to obtain a  $k$ -connected graph. However,

there is an example such that the number of edges added by this procedure cannot be bounded by  $O(\delta k)$  over the optimum.

In this paper, for arbitrary  $k \geq 4$  and  $\ell = k - \delta$ , we consider the problem of augmenting an  $\ell$ -connected graph  $G$  by adding the smallest number of new edges in order to make  $G$   $k$ -connected. We first present a lower bound on the number of edges that is necessary to make a given graph  $G$   $k$ -connected, and then show that the lower bound plus  $\delta(k-1) + \max\{0, (\ell-1)(\ell-3)-1\}$  edges suffices. The task of constructing such set of new edges can be done in  $O(\delta(k^2n^2 + k^3n^{3/2}))$  time.

The paper is organized as follows. In Section 2 we state our main result that  $k$ -VCAP is approximable within the absolute error  $O(\delta k)$  for a  $(k-\delta)$ -connected graph, after introducing some basic notations and deriving two lower bounds on the optimal value of the problem. In Section 3, we describe an outline of our approximation algorithm, called V-AUGMENT, for  $k$ -VCAP. In Section 4, we describe that the first lower bound can be computed in polynomial time. After stating several properties of  $k$ -connected graphs in Section 5, we show in Section 6, some previously known and newly derived edge-splitting operations (which are procedures for replacing two edges with a single edge while preserving  $k$ -connectivity). In Sections 7 and 8, we prove the correctness of V-AUGMENT. In Section 9 we state some concluding remarks.

## 2 Main Theorem

Let  $G = (V, E)$  stand for an undirected graph with a set  $V$  of *vertices* and a set  $E$  of *edges*, where we denote  $|V|$  by  $n$  (or by  $n(G)$ ) and  $|E|$  by  $m$  (or by  $m(G)$ ). An edge with end vertices  $u$  and  $v$  is denoted by  $(u, v)$ . In  $G = (V, E)$ , its vertex set  $V$  and edge set  $E$  may be denoted by  $V(G)$  and  $E(G)$ , respectively. A singleton set  $\{x\}$  may be simply written as  $x$ . For a subset  $V' \subseteq V$  (resp.,  $E' \subseteq E$ ) in  $G$ ,  $G[V']$  (resp.,  $G[E']$ ) denotes the subgraph induced by  $V'$  (resp.,  $G[E'] = (V, E')$ ). For  $V' \subset V$  (resp.,  $E' \subseteq E$ ), we denote subgraph  $G[V - V']$  (resp.,  $G[E - E']$ ) also by  $G - V'$  (resp.,  $G - E'$ ). For  $E' \subset E$ , we denote  $V(G[E'])$  by  $V[E']$ . For an edge set  $E'$  with  $E' \cap E = \emptyset$ , we denote the augmented graph  $G = (V, E \cup E')$  by  $G + E'$ . For two disjoint subsets of vertices  $X, Y \subset V$ , we denote by  $E_G(X, Y)$  the set of edges  $e = (x, y)$  such that  $x \in X$  and  $y \in Y$ , and also denote  $|E_G(X, Y)|$  by  $c_G(X, Y)$ . In particular,  $E_G(u, v)$  is the set of edges with end vertices  $u$  and  $v$ . A *partition*  $X_1, \dots, X_t$  of the vertex set  $V$  means a family of nonempty disjoint subsets of  $V$  whose union is  $V$ , and a *subpartition* of  $V$  means a partition of a subset  $V'$  of  $V$ . For a subset  $X$  of  $V$ , a vertex  $v \in V - X$  is called a *neighbor* of  $X$  if it is adjacent to some vertex  $u \in X$ , and the set of all neighbors of  $X$  is denoted by  $\Gamma_G(X)$ . A maximal connected subgraph  $G'$  in a graph  $G$  is called a *component* of  $G$  (for notational convenience, a component  $H$  may be represented by its vertex set  $X = V(H)$ ), and denote the set of all components in  $G$  by  $\mathcal{C}(G)$  and the number of components in  $G$  by  $p(G)$ . A *disconnecting set* of  $G$  is defined as a subset  $S$  of  $V$  such that  $p(G - S) > p(G)$  holds and no  $S' \subset S$  has this property. The *local vertex-connectivity*  $\kappa_G(x, y)$  for two vertices  $x, y \in V$  is defined to be the number of internally-disjoint paths between  $x$  and  $y$  in  $G$ . By Menger's theorem,  $\kappa_G(x, y)$  for nonadjacent vertices  $x$  and  $y$  is equal to the minimum size of a disconnecting set that separates  $x$  and  $y$ . A component  $G'$  of  $G$  with  $|V(G')| \geq 3$  always has a disconnecting set unless  $G$  is a complete graph  $K_n$ . For a connected  $G$ , a disconnecting set of the minimum size is called a *minimum disconnecting set*, and its size, denoted by  $\kappa(G)$ ,

is called the *vertex-connectivity* of  $G$ ; we define  $\kappa(G) = 0$  if  $G$  is not connected, and  $\kappa(G) = n - 1$  if  $G$  is a complete graph  $K_n$ . A graph  $G$  is called  $k$ -connected if  $\kappa(G) \geq k$ . A subset  $T \subset V$  is called *tight* if  $\Gamma_G(T)$  is a minimum disconnecting set in  $G$ . A tight set  $D$  is called *minimal* if no proper subset  $D'$  of  $D$  is tight (hence a minimal tight set  $D$  induces a connected subgraph  $G[D]$ ). We denote a family of all minimal tight sets in  $G$  by  $\mathcal{D}(G)$ . Let  $t(G)$  be the maximum number of pairwise disjoint minimal tight sets in  $G$ , and let  $\beta(G) = \max\{p(G - S) | S \text{ is a minimum disconnecting set in } G\}$  (hence  $|\mathcal{D}(G)| \geq t(G) \geq \beta(G)$ ).

For an initial graph  $G$  and a fixed integer  $k \geq 1$ , let  $opt_k(G)$  denote the optimal value of the  $k$ -VCAP in  $G$ , i.e., the minimum size  $|E'|$  of a set  $E'$  of new edges to obtain an  $k$ -connected graph  $G + E'$ . Several algorithms have been developed for  $k$ -VCAP in the case where an initial graph  $G$  is  $(k - 1)$ -connected. These algorithms use the following lower bound on  $opt_k(G)$ . If  $\kappa(G) = k - 1$ , then we easily observe that  $\mathcal{M}(G) = \max\{\lceil t(G)/2 \rceil, \beta(G) - 1\}$  is a lower bound on  $opt_k(G)$ . Eswaran and Tarjan [2] proved that 2-VCAP can be solved by finding a set of  $\mathcal{M}(G)$  edges. Watanabe and Nakamura [17] stated the same result for 3-VCAP. Thus  $\mathcal{M}(G)$  is indeed the optimal value for  $\kappa(G) = k - 1$  and  $k = 2, 3$ , while it is known that  $\mathcal{M}(G)$  can be smaller than the optimal value for general  $k \geq 4$ . It is reported in [5] that 4-VCAP can be solved in polynomial time for an arbitrary initial graph  $G$ . For  $k \geq 5$ , Jordán proved [10, 11] that  $k$ -VCAP with  $\kappa(G) = k - 1$  can be solved by an approximation algorithm which finds a solution with absolute error at most  $(k - 2)/2$ .

In what follows, we derive two types of lower bounds,  $\alpha_k(G)$  and  $\beta_k(G) - 1$ , on  $opt_k(G)$ , where  $\kappa(G)$  is not necessarily  $k - 1$ .

We call a subset  $X \subseteq V$  *dominating* in  $G$  if  $V - X - \Gamma_G(X) = \emptyset$ , and *non-dominating* if  $V - X - \Gamma_G(X) \neq \emptyset$ .

To make  $G$   $k$ -connected, it is necessary to add at least  $\max\{k - |\Gamma_G(X)|, 0\}$  edges between  $X$  and  $V - X - \Gamma_G(X)$  for any non-dominating set  $X \subset V$ . Given a family  $\mathcal{X} = \{X_1, \dots, X_p\}$  of disjoint non-dominating sets, the total sum  $\sum_{i=1, \dots, p} \max\{k - |\Gamma_G(X_i)|, 0\}$  of “deficiencies” over  $\mathcal{X}$  is decreased by at most two by adding one new edge to  $G$ . From this, we need at least  $\lceil \alpha_k(G)/2 \rceil$  new edges to make  $G$   $k$ -connected, where

$$\alpha_k(G) = \max_{\substack{\text{all families } \mathcal{X} \text{ of disjoint} \\ \text{non-dominating sets}}} \left\{ \sum_{X \in \mathcal{X}} (k - |\Gamma_G(X)|) \right\}. \quad (1)$$

(Note that  $t(G) = \alpha_k(G)$  holds if  $\kappa(G) = k - 1$ .)

We now consider another case in which new edges becomes necessary. For a vertex subset  $S \subseteq V$  of  $G$  with  $|S| = k - 1$ , let  $T_1, \dots, T_q$  denote all the components in  $G - S$ , where  $q = p(G - S)$ . To make  $G$   $k$ -connected, a new edge set  $E'$  must be added to  $G$  so that all  $T_i$  form a single connected component in  $(G + E') - S$ . For this, it is necessary to add at least  $p(G - S) - 1$  edges to connect all components in  $G - S$ , where  $S$  is not necessarily a minimum disconnecting set of  $G$  if  $\kappa(G) < k - 1$ . Here we define

$$\beta_k(G) = \max_{\substack{\text{all } S \subseteq V \text{ with } |S| = k - 1}} \left\{ p(G - S) \right\}. \quad (2)$$

Thus at least  $\beta_k(G) - 1$  new edges are necessary to make  $G$   $k$ -connected. Define

$$\gamma_k(G) = \max\{\lceil \alpha_k(G)/2 \rceil, \beta_k(G) - 1\}.$$

The next lemma combines the above two lower bounds.

**Lemma 1. (Lower Bound)** *For a given graph  $G$ , it holds  $\gamma_k(G) \leq \text{opt}_k(G)$ .*  $\square$

In this paper, we prove the next result.

**Theorem 1.** *Let  $G$  be an  $\ell$ -connected graph with  $\ell \geq 0$ . Then, for any integers  $k \geq 4$  and  $\delta = k - \ell \geq 2$ , it holds*

$$\text{opt}_k(G) \leq \gamma_k(G) + \delta(k - 1) + \max\{0, (\delta - 1)(\ell - 3) - 1\},$$

and a feasible solution  $E'$  to  $k$ -VCAP with  $|E'| \leq \gamma_k(G) + \delta(k - 1) + \max\{0, (\delta - 1)(\ell - 3) - 1\}$  can be found in  $O(\delta(k^2 n^2 + k^3 n^{3/2}))$  time, where  $n = |V(G)|$ .  $\square$

### 3 Outline of Algorithm

In this section, we give a sketch of our algorithm for finding a set  $E'$  of new edges in Theorem 1, where the algorithm also plays a role proving the theorem.

#### 3.1 $s$ -Basal $k$ -Connectivity

A graph  $H$  with a designated vertex  $s \in V(H)$ , where  $H - s$  is denoted by  $G$ , is called  $s$ -basally  $k$ -connected if

$$|\Gamma_G(X)| + |\Gamma_H(s) \cap X| \geq k \quad \text{for all non-dominating sets } X \subset V(G) \text{ in } G \quad (3)$$

with  $|\Gamma_G(X)| + |X| \geq k$ ,

$$|\Gamma_G(x)| + c_H(s, x) \geq k \quad \text{for all singleton sets } X = \{x\} \subset V(G). \quad (4)$$

*Claim.* For an  $s$ -basally  $k$ -connected graph  $H$ , it holds

$$|\Gamma_G(X)| + c_H(s, X) \geq k \quad \text{for all non-dominating sets } X \subset V \text{ in } G = H - s. \quad (5)$$

#### 3.2 Edge-Splitting Operation

An *edge-splitting* operation is defined as follows. Given a graph  $H$  with a designated vertex  $s$  and vertices  $u, v \in \Gamma_H(s)$  (possibly  $u = v$ ), we construct graph  $H'$  from  $H$  by deleting one edge from each of  $E_H(s, u)$  and  $E_H(s, v)$ , and adding new one edge to  $E_H(u, v)$ :  $c_{H'}(s, u) := c_H(s, u) - 1$ ,  $c_{H'}(s, v) := c_H(s, v) - 1$ ,  $c_{H'}(u, v) := c_H(u, v) + 1$ , and  $c_{H'}(x, y) := c_H(x, y)$  for all other pairs  $x, y \in V(H) - s$ . In the case of  $u = v$ , we interpret that  $c_{H'}(s, u) := c_H(s, u) - 2$ ,  $c_{H'}(u, u) := c_H(u, u) + 1$ , and  $c_{H'}(x, y) := c_H(x, y)$  for all other pairs  $x, y \in V$ . We say that  $H'$  is obtained from  $H$  by *splitting* a pair of edges  $(s, u)$  and  $(s, v)$  (or by splitting  $(s, u)$  and  $(s, v)$ ). Conversely, we say that  $H'$  is obtained from  $H$  by *hooking up* an edge  $(u, v) \in E(H - s)$  at  $s$ , if we construct  $H'$  by replacing an edge  $(u, v)$  with two edges  $(s, u)$  and  $(s, v)$  in  $H$ .

### 3.3 Entire Algorithm

Given a graph  $G = (V, E)$ , we try to start with computing the lower bound  $\alpha_k(G)$ . For this, we add a new vertex  $s$  to  $G$  together with some new edges between  $s$  and  $V$  such that each non-dominating set  $X$  in  $G$  with  $\Gamma_G(X) < k$  receives at least  $k - \Gamma_G(X)$  edges between  $s$  and  $X$ , where multiple edges are allowed between  $s$  and a vertex  $v \in V$ . In the resulting graph  $H^*$ , we then split edges at  $s$  so that the vertex-connectivity of  $H^* - s$  increases one by one. During the algorithm, we may further add to  $H^* - s$  some new edges (which are not generated by splitting at  $s$ ).

#### Algorithm V-AUGMENT

**Input:** An undirected graph  $G = (V, E)$  and integers  $k \geq 4$  and  $\ell \geq 0$  such that  $|V| \geq k + 1$ ,  $\kappa(G) = \ell$  and  $k - \ell \geq 2$ .

**Output:** A set  $E'$  of new edges with  $|E'| \leq \text{opt}_k(G) + \delta(k-1) + \max\{0, (\delta-1)(\ell-3)-1\}$  such that  $G^* = G + E'$  is  $k$ -connected, where  $\delta = k - \ell$ .

**Step I (Addition of vertex  $s$  and associated edges):** Add a new vertex  $s$  together with a set  $F^*$  of edges between  $s$  and  $V$  such that the resulting graph  $H^* = (V \cup \{s\}, E \cup F^*)$  is  $s$ -basally  $k$ -connected and  $F^*$  is minimal subject to this property.

*Property 1.* (1) The graph  $H^*$  can be computed in  $O(\min\{k, \sqrt{n}\}kn^2)$  time.

(2) If  $|\Gamma_{H^*}(s)| \leq k$ , then there exists a set  $E'$  of at most  $\delta(k-1)$  new edges such that  $G + E'$  is  $k$ -connected, and such  $E'$  can be found in  $O(\min\{k, \sqrt{n}\}((\delta-1)k^3n + \delta kn^2))$  time.

(3) If  $|\Gamma_{H^*}(s)| \geq k + 1$ , then  $|F^*| = \alpha_k(G)$  holds.  $\square$

Based on this property, if  $|\Gamma_{H^*}(s)| \leq k$ , then we find an  $E'$  in Property 1(2) and halt; we proceed to Step II otherwise.

**Step II (Increasing the vertex-connectivity from  $\ell$  to  $k-1$ ):** Let  $j := \ell$  and  $H_\ell := H^*$ .

For  $j = \ell, \dots, k-2$ , we repeat computing from  $H_j$  a graph  $H_{j+1}$  in the next property.

*Property 2.* For an  $s$ -basally  $k$ -connected graph  $H_j$  with  $\kappa(H_j - s) = j$ , there exists an  $s$ -basally  $k$ -connected graph  $H_{j+1}$  with  $\kappa(H_{j+1} - s) = j + 1$  such that  $H_{j+1}$  is constructed from  $H_j$  by splitting some edges incident to  $s$  and by adding a set  $\tilde{E}_j$  of at most  $\max\{2j-2, j+1\}$  new edges. Moreover, such  $H_{j+1}$  can be computed in  $O(\min\{k, \sqrt{n}\}(k^3n + kn^2))$  time.  $\square$

Thus, we obtain an  $s$ -basally  $k$ -connected graph  $H_{k-1} = (V \cup \{s\}, E \cup F_{k-1}^* \cup E_{k-1}^* \cup E_{k-1})$  with  $\kappa(H_{k-1} - s) = k - 1$  and  $F_{k-1}^* \subseteq F^*$ , where  $E_{k-1}^*$  is the set of edges generated by splitting edges in  $F^* - F_{k-1}^*$  at  $s$ , and  $E_{k-1} = E(H_{k-1} - s) - E - E_{k-1}^*$  with  $|E_{k-1}| \leq \sum_{i=\ell}^{k-2} \max\{2i-2, i+1\}$  (hence  $E_{k-1}$  is the set of edges directly added to  $H_{k-1}$ ).

**Step III (Increasing the vertex-connectivity from  $k-1$  to  $k$ ):** From  $H_{k-1}$  obtained in Step II, we compute an  $s$ -basally  $k$ -connected graph  $H_k = (V \cup \{s\}, E \cup F_k^* \cup E_k^* \cup E_k)$  in the next property.

*Property 3.* For an  $s$ -basally  $k$ -connected graph  $H_{k-1}$  obtained in Step II, there exists an  $s$ -basally  $k$ -connected graph  $H_k = (V \cup \{s\}, E \cup F_k^* \cup E_k^* \cup E_k)$  with  $F_k^* \subseteq F^*$  and  $E_k = E(H_k - s) - E - E_k^*$ , where  $E_k^*$  is the set of edges generated by splitting edges in  $F^* - F_k^*$  at  $s$ , such that  $|E_k^* \cup E_k| \leq |F^* - F_k^*|/2 + \sum_{i=\ell}^{k-2} \max\{2i-2, i+1\}$  and  $G_k = G + (E_k^* \cup E_k)$  ( $= H_k - s$ ) can be made  $k$ -connected by adding a set  $\hat{E}_k$  of at most  $\beta(G) - |E_k^* \cup E_k| - 1$  or  $t(G_k) - 1$  ( $\leq \max\{2k-3, k+1\} - 1$ ) new edges. Moreover, such  $H_k$  and  $\hat{E}_k$  can be found in  $O(\min\{k, \sqrt{n}\}(k^3n + kn^2))$  time.  $\square$

Then we augment  $G$  to a  $k$ -connected graph by adding with edge set  $E' = E_k^* \cup E_k \cup \hat{E}_k$ , where  $|E'| \leq \beta(G) - 1$  or  $|E'| \leq |F^* - F_k^*|/2 + \sum_{i=\ell}^{k-2} \max\{2i-2, i+1\} + t(G_k) - 1$  holds. If  $|E'| \leq \beta(G) - 1$  ( $\leq \text{opt}_k(G)$ ), then  $G + E'$  is an optimally augmented graph. If  $|E'| \leq |F^* - F_k^*|/2 + \sum_{i=\ell}^{k-2} \max\{2i-2, i+1\} + t(G_k) - 1$ , then by  $t(G_k) \leq |F_k^*|$  and  $|F^*| = \alpha_k(G)$ , we have  $|F^* - F_k^*|/2 + \sum_{i=\ell}^{k-2} \max\{2i-2, i+1\} + t(G_k) - 1 \leq [\alpha_k(G)/2] + [t(G_k)/2] - 1 + \sum_{i=\ell}^{k-2} \max\{2i-2, i+1\} \leq \text{opt}_k(G) + k - 2 + \sum_{i=\ell}^{k-2} \max\{2i-2, i+1\} \leq \text{opt}_k(G) + \delta(k-1) + \max\{0, (\delta-1)(\ell-3)-1\}$ .  $\square$

By summing up the running time in Steps I, II and III (where we apply Property 2 at most  $\delta$  times), the entire time complexity of V-AUGMENT is  $O(\delta \min\{k, \sqrt{n}\}(kn^2 + k^3n)) = O(\delta(k^2n^2 + k^3n^{3/2}))$ .

**Remark:** In general, it seems difficult to solve the maximization problem in 2 to compute  $\beta_k(G)$  in polynomial time. However, from our algorithm, if  $[\alpha_k(G)/2] + \delta(k-1) + \max\{0, (\delta-1)(\ell-3)-1\} + 1 < \beta_k(G)$ , then  $\beta_k(G)$  can be computed in polynomial time.  $\square$

## 4 Correctness of Step I

In this section, we observe the correctness of Step 1. For this, it suffices to prove Property 1.

We say that a disconnecting set  $S \subset V$  disconnects two disjoint subsets  $Y$  and  $Y'$  of  $V - S$  if no two vertices  $x \in Y$  and  $y \in Y'$  are connected in  $G - S$ . In particular, a disconnecting set  $S$  disconnects vertices  $x$  and  $y$  in  $V - S$  if  $x$  and  $y$  are contained in different components of  $G - S$ . A vertex subset  $X$  intersects another vertex subset  $Y$  if none of subsets  $X \cap Y$ ,  $X - Y$  and  $Y - X$  is empty. The following property holds for two vertex subsets  $X$  and  $Y$  in  $G = (V, E)$ :

$$|\Gamma_G(X)| + |\Gamma_G(Y)| \geq |\Gamma_G(X \cap Y)| + |\Gamma_G(X \cup Y)|. \quad (6)$$

**Proof of Property 1(1):** We start with the graph  $H$  obtained from  $G$  by adding a new vertex  $s$  and  $\max\{1, k - |\Gamma_G(v)|\}$  edges between  $s$  and each vertex  $v \in V$ . It is not difficult to see that  $H$  is  $s$ -basally  $k$ -connected. We then can check whether  $H - (s, v)$  remains  $s$ -basally  $k$ -connected or not for each vertex  $v \in V$  with  $c_H(s, v) = 1 > k - |\Gamma_G(v)|$  by computing  $\kappa_{H-(s,v)}(s, v)$ . Hence, the  $s$ -basal  $k$ -connectivity of  $H - (s, v)$  can be tested in  $O(m + \min\{k, \sqrt{n}\}kn)$  time by using the network flow computation 3 on a sparse spanning subgraph of  $H$  with  $O(kn)$  edges, where such sparsification

takes  $O(m)$  time [15, 16]. Since there are  $O(n)$  such computation of  $\kappa_{H^*}(s, v)$ , the total time complexity for computing  $H^*$  is  $O(\min\{k, \sqrt{n}\}kn^2)$ .  $\square$

**Proof of Property II(2):** To prove this property, we use the next lemma.

**Lemma 2.** [14] Let  $C$  be a cycle in a  $k$ -connected graph  $G = (V, E)$  such that  $\kappa(G - e) = k - 1$  holds for every  $e \in E(C)$ . Then there exists a vertex  $v \in V(C)$  with  $|\Gamma_G(v)| = k$ .  $\square$

Assume that  $|\Gamma_{H^*}(s)| \leq k$ . We start from  $G_\ell := G$  and continue to construct a  $(j+1)$ -connected graph  $G_{j+1}$  from  $G_j$  by adding a set  $E_j$  of new edges in the following way until  $j = k-1$  holds.

For each  $j \in \{\ell, \dots, k-1\}$ , since  $t(G_j) \leq |\Gamma_{H^*}(s)| \leq k$  holds,  $G_j$  can be made  $(j+1)$ -connected by adding a set  $E_j$  of new edges with  $|E_j| \leq k-1$  from Lemma 2. Note that  $G_{j+1} := G_j + E_j$  satisfies  $\kappa(G_{j+1}) = j+1$  and  $t(G_{j+1}) \leq k$  since  $(G_{j+1} \cup \{s\}) + F^*$  remains  $s$ -basally  $k$ -connected.

Consequently, we can find a solution  $\bigcup_{i=\ell}^{k-1} E_i$  whose size is  $|\bigcup_{i=\ell}^{k-1} E_i| \leq (k-\ell)(k-1) = \delta(k-1)$ .  $\square$

**Proof of Property II(3):** The minimality of  $F^*$  implies that for each edge  $(s, v) \in E_{H^*}(s, V) = F^*$ , there is a set  $X_v \subseteq V$  with  $v \in X_v$  such that

- (i)  $|\Gamma_G(X_v)| = k-1$ ,  $c_{H^*}(s, X_v) = c_{H^*}(s, v) = 1$ , and  $V - X_v - \Gamma_G(X_v) \neq \emptyset$ , or
- (ii)  $|X_v| = 1$  and  $|\Gamma_G(v)| + c_{H^*}(s, v) = k$ ,

and no proper subset  $X' \subset X_v$  satisfies this property. For an integer  $i \in \{0, 1, \dots, k-1\}$ , we call a set  $T \subseteq V$   $i$ -critical in  $H^*$ , if  $T$  satisfies  $V - T - \Gamma_G(T) \neq \emptyset$ ,  $|\Gamma_G(T)| = i$ ,  $|\Gamma_G(T)| + |\Gamma_{H^*}(s) \cap T| = k$ , and  $c_{H^*}(s, u) = 1$  for each  $u \in \Gamma_{H^*}(s) \cap T$ , and no proper subset  $T' \subset T$  satisfies this property for the fixed  $i$ . Note that  $c_{H^*}(s, T) = |\Gamma_{H^*}(s) \cap T| = k-i$  holds for an  $i$ -critical set  $T$  with  $i \in \{0, 1, \dots, k-1\}$ . We call a singleton set  $T = \{v\}$  with  $v \in V$   $k$ -critical if  $|\Gamma_G(v)| + c_{H^*}(s, v) = k$ , and  $c_{H^*}(s, v) > 0$  hold. Note that the above set  $X_v$  satisfying (i) (resp., (ii)) is  $(k-1)$ -critical (resp.,  $k$ -critical). Thus,

**Lemma 3.** Each  $v \in \Gamma_{H^*}(s)$  is contained in a  $(k-1)$ -critical set or a  $k$ -critical set.  $\square$

By using (II), we can prove the next property.

**Lemma 4.** Assume that  $H^*$  has an  $i$ -critical set  $T_i$  and a  $j$ -critical set  $T_j$  such that  $T_i$  and  $T_j$  intersect each other in  $G$ . If  $|\Gamma_{H^*}(s)| \geq k+1$ , then  $T_i \cup T_j$  contains an  $(i+j-h)$ -critical set  $T$  with  $\Gamma_{H^*}(s) \cap (T_i \cup T_j) \subseteq T$ , where  $h = |\Gamma_G(T_i \cap T_j)|$ .  $\square$

Let  $\mathcal{T}$  be a family of  $i$ -critical sets  $T \subset V$ ,  $0 \leq i \leq k$ , such that  $\Gamma_{H^*}(s) \subseteq \bigcup_{T \in \mathcal{T}} T$  holds and  $|\mathcal{T}|$  is the minimum; Lemma 3 says that such  $\mathcal{T}$  exists. Then we can observe by Lemma 4 that if  $|\Gamma_{H^*}(s)| \geq k+1$ , then every two sets in  $\mathcal{T}$  are pairwise disjoint. By  $|\Gamma_{H^*}(s)| \geq k+1$ , for a minimum family  $\mathcal{T}$  of  $i$ -critical sets with  $\Gamma_{H^*}(s) \subseteq \bigcup_{T \in \mathcal{T}} T$ , we have  $|F^*| = c_{H^*}(s, V) = \sum_{T \in \mathcal{T}} c_{H^*}(s, T) = \sum_{T \in \mathcal{T}} (k - |\Gamma_G(T)|) \leq \alpha_k(G)$ . Moreover, if  $|F^*| < \alpha_k(G)$ , then it is not difficult to see that at least one set  $X \in \mathcal{T}$  (or  $\{x\} \in \mathcal{T}$ ) would violate (3) or (4). Hence  $|F^*| \geq \alpha_k(G)$  also holds.  $\square$

## 5 Structure of $k$ -Connected Graphs

Before proving the correctness of Steps II and III of V-AUGMENT, we review some properties of a  $k$ -connected graph, which will be a basis for deriving edge-splitting operations in these steps.

**Lemma 5.** [10] *Let  $G$  be  $k$ -connected. If  $t(G) \geq k + 1$ , then any two minimal tight sets  $X, Y \in \mathcal{D}(G)$  are pairwise disjoint (i.e.,  $t(G) = |\mathcal{D}(G)|$ ).  $\square$*

For a subset  $S \subset V$  in  $G$ , we call the components in  $G - S$  the  $S$ -components. Note that the vertex set  $S$  is a disconnecting set in a connected  $G$  if and only if  $p(G - S) \geq 2$ . A tight set  $T$  is called a *superleaf*, if  $T$  contains exactly one minimal tight set  $D \in \mathcal{D}(G)$  and no superset  $T' \supset T$  satisfies this property. The following lemmas summarize some properties of superleaves.

**Lemma 6.** [1] *Let  $G = (V, E)$  be a connected graph with  $t(G) \geq \kappa(G) + 3$ .*

- (1) *For every minimal tight set, as well as every superleaf, the induced subgraph is connected.*
- (2) *For each minimal tight set  $D \in \mathcal{D}(G)$ , there is a unique superleaf  $Q$  containing  $D$ .*
- (3) *Every two superleaves are pairwise disjoint. Hence, a superleaf  $Q$  is disjoint from all other minimal tight sets in  $\mathcal{D}(G)$ , except for the one in  $\mathcal{D}(G)$  contained in  $Q$ .  $\square$*

We call a disconnecting set  $S$  a *shredder* if  $p(G - S) \geq 3$ .

**Lemma 7.** [89] *Let  $G = (V, E)$  satisfy  $t(G) \geq \kappa(G) + 3$ , and  $S$  be a shredder with  $|S| = \kappa(G)$ . If an  $S$ -component  $T \in \mathcal{C}(G - S)$  contains a minimal tight set  $D \in \mathcal{D}(G)$ , but no other minimal tight set in  $\mathcal{D}(G) - D$ , then  $T$  is the superleaf with  $T \supseteq D$ .  $\square$*

**Lemma 8.** [89] *Let  $S$  be a shredder with  $|S| = \kappa(G)$  in a connected graph  $G = (V, E)$ . If  $p(G - S) \geq \kappa(G) + 1$ , then every superleaf  $Q$  in  $G$  satisfies  $Q \cap S = \emptyset$ .  $\square$*

**Theorem 2.** [1, Lemma 5.8] *Let  $k \geq 2$  and  $G = (V, E)$  be a  $(k - 1)$ -connected graph such that  $t(G) \geq \max\{2k - 2, k + 2\}$  and  $\beta(G) \leq \lceil t(G)/2 \rceil$ . Suppose that  $G$  has a shredder  $S$  with  $|S| = \kappa(G)$  such that every  $S$ -component contains exactly one minimal tight set. Then  $t(G) = 2k - 2$  holds and the minimum number of edges required to make  $G$   $k$ -connected is  $2k - 4$  if  $G$  is a complete bipartite graph  $K_{k-1, k-1}$ , and  $k - 2 + \lceil (k - 1)/2 \rceil$  otherwise. Moreover, such set of edges can be found in  $O(n)$  time if all minimal tight sets in  $G$  have been found.  $\square$*

We show a new property of a shredder in a  $k$ -connected graph (the proof is omitted).

**Lemma 9.** *Let  $S$  be a shredder with  $|S| = \kappa(G)$  in a connected graph  $G = (V, E)$ . Assume that every  $S$ -component is a superleaf in  $G$ .*

- (1) *If  $p(G - S) \geq \kappa(G) + 1$ , then every minimum disconnecting set  $S_1$  other than  $S$  satisfies  $p(G - S_1) = 2$  and  $S_1 \cap D = \emptyset$  for all  $D \in \mathcal{D}(G)$ , and has an  $S_1$ -component  $T_1 \subseteq Q$  for some  $S$ -component  $Q \in \mathcal{C}(G - S)$ .*
- (2) *If  $p(G - S) \geq \kappa(G) + 2$ , then for any subset  $W \subset V$  with  $|W| = \kappa(G) + 1$  and  $p(G - W) \geq 2$ , there is at most one minimal tight set  $D \in \mathcal{D}(G)$  with  $D \cap W \neq \emptyset$ .  $\square$*

## 6 Edge-Splitting Preserving $k$ -Connectivity

### 6.1 Edge-Splitting in $(k - 1)$ -Connected Graphs

Let  $H = (V \cup \{s\}, E)$  be a graph with a designated vertex  $s$  and  $|V| \geq k + 1$  such that  $\kappa_H(x, y) \geq k$  holds for all distinct two vertices  $x, y \in V$  (which is equivalent to the condition that  $|\Gamma_H(Y)| \geq k$  for all non-dominating sets  $Y \subseteq V$  in  $G$ ). Let  $G = H - s$ , and assume that  $G$  is not  $k$ -connected. Then  $\kappa(G) = k - 1$  holds, since  $G$  satisfies  $\kappa_G(x, y) \geq \kappa_H(x, y) - 1 \geq k - 1$  for all  $x, y \in V$ . Note that  $\Gamma_H(s) \cap D \neq \emptyset$  holds for every minimal tight set  $D \in \mathcal{D}(G)$  since otherwise  $\kappa_H(x, y) \geq k$  cannot hold for  $x \in D$  and  $y \in V - D - \Gamma_G(D)$ . A pair  $\{(s, u), (s, v)\}$  of two edges in  $E_H(s)$  is called  *$k$ -splittable*, if the graph  $H'$  resulting from splitting edges  $(s, u)$  and  $(s, v)$  satisfies  $\kappa_{H'}(x, y) \geq k$  for all pairs  $x, y \in V$ .

The following theorems describe some conditions that admit  $k$ -splittable splittings at  $s$  in a  $(k - 1)$ -connected graph  $G$ .

**Theorem 3.** [1] *Let  $H = (V \cup \{s\}, E)$  be a graph with a designated vertex  $s$ ,  $k \geq 2$  be an integer such that  $\kappa_H(x, y) \geq k$  for all pairs  $x, y \in V$ , and let  $G = H - s$  satisfy  $\kappa(G) = k - 1$  and  $t(G) \geq k + 2$ . Assume that  $G$  has three distinct superleaves  $Q_1, Q_2$  and  $Q_3$  such that  $\Gamma_G(Q_1) \cap Q_2 = \emptyset = \Gamma_G(Q_1) \cap Q_3$  holds and  $\Gamma_G(Q_1)$  is not a shredder in  $G$ . Let  $D_i \subseteq Q_i$ ,  $i = 1, 2, 3$ , be the minimal tight set in  $\mathcal{D}(G)$ . Then, for any three vertices  $x_i \in \Gamma_H(s) \cap D_i$ ,  $i = 1, 2, 3$ , at least one of  $\{(s, x_1), (s, x_2)\}$ ,  $\{(s, x_2), (s, x_3)\}$  and  $\{(s, x_3), (s, x_1)\}$  is  $k$ -splittable. Moreover  $t(H' - s) = t(G) - 2$  holds for the resulting graph  $H'$  from the splitting.*  $\square$

**Theorem 4.** [89] *Let  $H = (V \cup \{s\}, E)$  be a graph with a designated vertex  $s$ ,  $k \geq 2$  be an integer such that  $\kappa_H(x, y) \geq k$  for all pairs  $x, y \in V$ , and let  $G = H - s$  satisfy  $\kappa(G) = k - 1$  and  $t(G) \geq \max\{2k - 2, k + 2\}$ . Let  $Q \subset V$  be an arbitrary superleaf such that  $S = \Gamma_G(Q)$  is a shredder in  $G$ . If  $G$  has a set  $T^* \in \mathcal{C}(G - S) - Q$  with  $c_H(s, T^*) \geq 2$ , then  $\{(s, x), (s, y)\}$  is  $k$ -splittable for any vertices  $x \in \Gamma_H(s) \cap Q$  and  $y \in \Gamma_H(s) \cap T^*$ .*  $\square$

Based on Theorems 2, 3 and 4, we can show the following three new properties on  $k$ -splittable splitting pairs (the proofs are omitted).

**Lemma 10.** *Let  $H = (V \cup \{s\}, E)$  be a graph with a designated vertex  $s$ ,  $k \geq 2$  be an integer such that  $\kappa_H(x, y) \geq k$  for all pairs  $x, y \in V$ , and let  $G = H - s$  satisfy  $\kappa(G) = k - 1$ ,  $t(G) \geq \max\{2k - 2, k + 2, \beta(G) + 1\}$ , and  $\beta(G) - 1 \geq \lceil t(G)/2 \rceil \geq k - 1$ . Then there is a  $k$ -splittable pair  $\{(s, x), (s, y)\}$  such that  $\beta(G + (x, y)) = \beta(G) - 1$ .*  $\square$

**Lemma 11.** *Let  $H = (V \cup \{s\}, E)$  be a graph with a designated vertex  $s$ ,  $k \geq 2$  be an integer such that  $\kappa_H(x, y) \geq k$  for all pairs  $x, y \in V$ , and let  $G = H - s$  satisfy  $\kappa(G) = k - 1$  and  $\beta(G) = t(G) \geq \max\{2k - 2, k + 2\}$ . Let  $S^*$  be a shredder in  $G$  with  $|S^*| = k - 1$  and  $p(G - S^*) = \beta(G)$ . Assume that  $G$  has an  $S^*$ -component  $T_1 \in \mathcal{C}(G - S^*)$  and an edge  $e = (v_1, v_2)$  such that  $v_1 \in T_1$ ,  $v_2 \in T_1 \cup S^*$ , and  $p(G - S^*) = p(G - e - S^*)$ . Let  $H_1 := H - e + \{(s, v_1), (s, v_2)\}$ . Then there is an edge-splitting of a pair  $\{(s, u_1), (s, u_2)\}$  at  $s$  in  $H_1$  such that the graphs  $H' = H_1 - \{(s, u_1), (s, u_2)\} + \{(u_1, u_2)\}$  and  $G' = H' - s$  satisfy  $\kappa_{H'}(x, y) \geq k$  for all pairs  $x, y \in V$ ,  $\kappa(G') = k - 1$  and  $\beta(G' - S^*) = \beta(G - S^*) - 1$ .*  $\square$

**Lemma 12.** Let  $H = (V \cup \{s\}, E)$  be a graph with a designated vertex  $s$ ,  $k \geq 2$  be an integer such that  $\kappa_H(x, y) \geq k$  for all pairs  $x, y \in V$ , and let  $G = H - s$  satisfy  $\kappa(G) = k - 1$ ,  $t(G) \geq \max\{2k - 2, k + 2\}$ , and  $\beta(G) \leq \lceil t(G)/2 \rceil$ . Then (a) there is a  $k$ -splittable pair  $\{(s, x), (s, y)\}$  such that  $t(G + (x, y)) \leq t(G) - 1$ , or (b)  $t(G) = 2k - 2$  holds and  $G$  can be made  $k$ -connected by adding at most  $2k - 4$  new edges.  $\square$

## 6.2 Edge-Splitting in an Arbitrary Graph

In this section, we consider an  $s$ -basally  $k$ -connected graph  $H = (V \cup \{s\}, E)$  with a designated vertex  $s$ . A pair  $\{(s, u), (s, v)\}$  of two edges in  $E_H(s)$  is called  $k$ -feasible at  $s$ , if the graph  $H'$  resulting from splitting edges  $(s, u)$  and  $(s, v)$  remains  $s$ -basally  $k$ -connected. We show two properties on a  $k$ -feasible edge-splitting at  $s$  in  $H$  (the proofs are omitted).

**Theorem 5.** Let  $H = (V \cup \{s\}, E)$  be an  $s$ -basally  $k$ -connected with a designated vertex  $s$ , and  $k_1$  be an integer with  $k > k_1 \geq 2$  such that  $G = H - s$  satisfies  $\kappa(G) = k_1 - 1$  and  $t(G) \geq k_1 + 2$ . Then there is a subgraph  $H_1 = (V \cup \{s\}, E_1 = E(G) \cup F')$  of  $H$  such that  $F' \subseteq E_H(s)$ ,  $|F'| = |\mathcal{D}(G)|$  and  $c_{H_1}(s, D) = 1$  holds for all  $D \in \mathcal{D}(G)$ . For such  $H_1$ , if a pair  $\{(s, v_1), (s, v_2)\}$  is  $k_1$ -splittable at  $s$  in  $H_1$ , then  $\{(s, v_1), (s, v_2)\}$  is  $k$ -feasible at  $s$  in  $H$ .  $\square$

**Theorem 6.** Let  $H = (V \cup \{s\}, E)$  be an  $s$ -basally  $k$ -connected graph with a designated vertex  $s$  and  $k_1$  be an integer with  $k > k_1 \geq 1$  such that  $G = H - s$  satisfies  $\kappa(G) = k_1 - 1$  and  $t(G) \geq k_1 + 2$ . Assume that  $G$  has a minimum disconnecting set  $S^*$  with  $p(G - S^*) \geq k_1 + 2$  such that every  $S^*$ -component  $Q_i \in \{Q_1, \dots, Q_p\} = \mathcal{C}(G - S^*)$  is a superleaf in  $G$ , where  $p = t(G) = p(G - S^*)$  holds (by Lemmas 7 and 8) and  $D_i \in \mathcal{D}(G)$  denotes the minimal tight set contained in  $Q_i$ . Then there is a subset  $F' = \{e_i, e'_i \mid i = 1, 2, \dots, p\} \subseteq E_H(s)$  such that for  $e_i = (s, u_i)$  and  $e'_i = (s, v_i)$ ,  $i = 1, 2, \dots, p$ ,  $u_i, v_i \in D_i$  holds. For such  $F'$ , the graph  $H' = H - \cup_{i=2}^p \{(s, v_{i-1}), (s, u_i)\} + \cup_{i=2}^p \{(v_{i-1}, u_i)\}$  is  $s$ -basally  $k$ -connected and  $G' = H' - s$  satisfies  $\kappa(G') = k_1$ .  $\square$

## 7 Correctness of Step II

To show the correctness of Step II, it suffices to prove Property 2. For this, we present an algorithm, called  $\kappa$ -SPLIT1, which finds from a given  $s$ -basally  $k$ -connected graph  $H$  with  $\kappa(H - s) \leq k - 2$  an  $s$ -basally  $k$ -connected graph  $H^*$  with  $\kappa(H^* - s) = \kappa(H - s) + 1$  by splitting some edges incident to  $s$  and by adding at most  $\max\{2\kappa(H - s) - 2, \kappa(H - s) + 1\}$  new edges.

**Algorithm  $\kappa$ -SPLIT1( $H, s, k$ )**

**Input:** An  $s$ -basally  $k$ -connected graph  $H = (V \cup \{s\}, E \cup F)$  with a designated vertex  $s$ ,  $F = E_H(s)$ , and  $|V| \geq k + 1$ . Let  $k_1 = \kappa(H - s) + 1$  and  $G = H - s$ .

**Output:** An  $s$ -basally  $k$ -connected graph  $H^* = (V \cup \{s\}, E \cup F_0 \cup E_1^* \cup E_2^*)$ , where

$F_0 \subseteq F$ ,  $E_1^*$  is the set of edges generated by splitting the edges in  $E_H(s) - F_0$  at  $s$ , and  $E_2^*$  is a set of new edges connecting vertices in  $V$ , such that  $\kappa(H^* - s) = k_1$  and  $|E_2^*| \leq \max\{2k_1 - 4, k_1\}$ .

**Step 1.** If  $k_1 = 1$ , then choose a subset  $F' = \{e_i = (s, u_i), e'_i = (s, v_i) \mid i = 1, 2, \dots, p\} \subseteq E_H(s)$  such that  $u_i, v_i \in D_i \in \{D_1, D_2, \dots, D_p\} = \mathcal{D}(G)$  holds for each  $i = 1, 2, \dots, p$ . Then the graph  $H^* := H + \cup_{i=2}^p \{(v_{i-1}, u_i)\} - \cup_{i=2}^p \{(s, v_{i-1}), (s, u_i)\}$  is  $s$ -basally  $k$ -connected and  $H^* - s$  is connected by Theorem 6. Halt outputting  $H^*$ , where  $E_2^* = \emptyset$ .

If  $k_1 \geq 2$ , then we go to Step 2 after setting  $H_2 := (V \cup \{s\}, E \cup F_2)$  with a minimal subset  $F_2 \subseteq F$  such that every  $D_i \in \mathcal{D}(G)$  satisfies  $c_{H_2}(s, D_i) \geq 1$ .

**Step 2.** Let  $G_2 := H_2 - s$ . If  $t(G_2) \leq \max\{2k_1 - 3, k_1 + 1\}$ , then  $G_2$  can be made  $k_1$ -connected by adding a set  $E_2^*$  of at most  $\max\{2k_1 - 4, k_1\}$  new edges by Lemma 2. Halt after outputting  $H^* := H_2 + (F - F_2) \cup E_2^*$ .

Otherwise, every  $D_i \in \mathcal{D}(G)$  satisfies  $c_{H_2}(s, D_i) = 1$  (by Lemma 5) and let  $\Gamma_{H_2}(s) \cap D_i = \{u_i\}$ . Repeat the following procedure (A) or (B) while  $t(G_2) \geq \max\{2k_1 - 2, k_1 + 2\}$ , where we execute procedure (A) if  $\beta(G_2) \leq \lceil t(G_2)/2 \rceil$  or procedure (B) otherwise.

#### Procedure (A):

- (A-1) If there is a  $k_1$ -splittable pair  $\{(s, u), (s, v)\}$  such that  $H' = H_2 - \{(s, u), (s, v)\} + \{(u, v)\}$  with  $G' = H' - s$  satisfies  $t(G') \leq t(G_2) - 1$ , then continue executing Step 2 after setting  $H_2 := H_2 - \{(s, u), (s, v)\} + \{(u, v)\}$ .
- (A-2) Otherwise, by Lemma 12,  $G_2$  can be made  $k_1$ -connected by adding a set  $E_2^*$  of at most  $2k_1 - 4$  new edges. Halt outputting  $H^* := H_2 + (F - F_2) \cup E_2^*$ .

*Property 4.* Each iteration of procedure (A-1) decreases  $t(G_2)$  by at least one.  $\square$

**Procedure (B):** Choose a minimum disconnecting set  $S^*$  in  $G_2$  satisfying  $p(G_2 - S^*) = \beta(G_2)$ , where  $S^*$  is a shredder in  $G_2$  (by  $\beta(G_2) \geq \lceil t(G_2)/2 \rceil + 1 \geq \max\{k_1, (k_1 + 2)/2 + 1\} \geq 3$ ).

- (B-1) If  $t(G_2) \geq \beta(G_2) + 1$ , then find a  $k_1$ -splittable pair  $\{(s, u), (s, v)\}$  in Lemma 10 such that  $H' = H_2 - \{(s, u), (s, v)\} + \{(u, v)\}$  with  $G' = H' - s$  satisfies  $\beta(G') \leq \beta(G_2) - 1$ . Continue executing Step 2 after setting  $H_2 := H_2 - \{(s, u), (s, v)\} + \{(u, v)\}$ .
- (B-2) Otherwise ( $\beta(G_2) = t(G_2)$ ), by Lemma 7 and  $t(G_2) \geq k_1 + 2$ , every  $S^*$ -component  $T \in \mathcal{C}(G_2 - S^*)$  is a superleaf in  $G_2$ . Moreover, Lemma 8 and  $p(G_2 - S^*) \geq k_1$  tell that every superleaf  $Q$  in  $G_2$  satisfies  $Q \cap S^* = \emptyset$ . Hence we have  $p(G_2 - S^*) = t(G_2) \geq k_1 + 2$ . Let  $\mathcal{D} := \{D \in \mathcal{D}(G) \mid c_{H_2}(s, D) = 1\}$  and  $(s, v_i) \in E_H(s, D_i) - \{(s, u_i)\}$  for  $D_i \in \mathcal{D}$  (such  $v_i$  exists since  $H$  satisfies 5 with respect to  $k \geq k_1 + 1$ ). Let  $H_3 := H + \cup_{i=2}^p \{(v_{i-1}, u_i)\} - \cup_{i=2}^p \{(s, v_{i-1}), (s, u_i)\}$ . Halt outputting  $H^* := H_3$ , where  $E_2^* = \emptyset$  and  $H_3$  remains  $s$ -basally  $k$ -connected by Theorem 6.

*Property 5.* Each iteration of (B-1) decreases  $\beta(G_2)$  by one.  $\square$

During execution of Step 2, no new edge is added to  $H_2$  immediately before constructing the final  $H^*$ . Then it is clear that the output  $H^* = (V \cup \{s\}, E \cup F_0 \cup E_1^* \cup E_2^*)$  is constructed from  $H$  by splitting edges in  $F - F_0$  (where  $E_1^*$  denotes the set of the resulting split edges) and adding edges in  $E_2^*$ . We prove that the output  $H^*$  is  $s$ -basally  $k$ -connected, and satisfies  $\kappa(H^* - s) = k_1$  and  $|E_2^*| \leq \max\{2k_1 - 4, k_1\}$ . If  $H^*$  is output in Step 1, then by Theorem 6 the output  $H^*$  is  $s$ -basally connected,

and satisfies  $\kappa(H^* - s) = k_1 = 1$  and  $E_2^* = \emptyset$ . In procedures (A) and (B), we see from Theorem 5 that  $H_2 \cup (F - F_2)$  is  $s$ -basally  $k$ -connected. In procedure (A), Lemma 12 ensures that we can execute either (A-1) or (A-2) if  $\beta(G_2) \leq \lceil t(G_2)/2 \rceil$ , and this proves Property 4. In procedure (B), we can execute either (B-1) or (B-2) by Lemma 10 and by Theorem 6, respectively, where Theorem 6 is applicable to (B-2) by Lemmas 7 and 8. Property 5 follows from Lemma 10. Therefore, by Properties 4 and 5,  $H_2$  satisfies  $t(G_2) \leq \max\{2k_1 - 3, k_1 + 1\}$ , or condition of (A-2) or (B-2) after  $t(H_2 - s) + \beta(H_2 - s) \leq n + k$  iterations of Step 2. If  $t(G_2) \leq \max\{2k_1 - 3, k_1 + 1\}$  holds in Step 2, then by Lemma 2 we can obtain a desired graph  $H^*$ .

The algorithm  $\kappa$ -SPLIT1( $H, s, k$ ) can be implemented to run in  $O(\min\{k_1, \sqrt{n}\} (k_1^3 n + k_1 n^2))$  time as follows. The running time of  $\kappa$ -SPLIT1( $H, s, k$ ) is equal to the complexity of finding a sequence of  $k_1$ -splittable pairs in  $H_2$ , plus the complexity of computing  $E_2^*$ . The complexity of finding a sequence of  $k_1$ -splittable pairs in  $H_2$  is equal to that of finding  $\kappa$ -splittable pairs in [8,9]. The edge set  $E_2^*$  can be computed in  $O(\min\{k_1, \sqrt{n}\} k_1^3 n)$  time by applying Phase 5 of Jordán's algorithm in [10].

## 8 Correctness of Step III

In this section, we prove the correctness of Step III by presenting an algorithm for computing a graph  $H_k$  in Property 3.

Given a graph  $H$  with a designated vertex  $s$ , we say that a graph  $H'$  is obtained from  $H$  by *shifting* an edge  $(s, u)$  to  $(s, v)$  at  $s$ , if we construct  $H'$  by replacing  $(s, u)$  with  $(s, v)$  in  $H$ .

**Algorithm  $\kappa$ -SPLIT2( $H, s, F, k$ )**

**Input:** An undirected graph  $H = (V \cup \{s\}, E \cup F_{k-1}^* \cup E_{k-1}^*)$  and an integer  $k \geq 2$  such that  $F_{k-1}^* = E_H(s)$  and  $\kappa(H - s) = k - 1$  hold, and  $\kappa_H(x, y) \geq k$  holds for every  $x, y \in V$ . Let  $H' = (V \cup \{s\}, E \cup F_{k-1}^* \cup F')$  denote the graph obtained from  $H$  by hooking up all edges in  $E_{k-1}^*$  at  $s$ , where  $F'$  is the set of the edges hooked up.

**Output:** An undirected graph  $H^* = (V \cup \{s\}, E \cup F_k^* \cup E_k^*)$  obtained from  $H'$  by splitting edges in  $F_{k-1}^* \cup F'$  and by shifting some split edges such that  $H^*$  satisfies  $\kappa_{H^*}(x, y) \geq k$ ,  $x, y \in V$  and  $G^* = H^* - s$  satisfies  $\kappa(G^*) = k - 1$  and one of the following (a) – (c), where  $E_{H^*}(s) = F_k^* \subseteq F_{k-1}^* \cup F'$ .

- (a)  $t(G^*) \leq \max\{2k - 3, k + 1\}$ .
- (b)  $t(G^*) = 2k - 2$  holds and  $G^*$  can be made  $k$ -connected by adding at most  $2k - 4$  new edges.
- (c) The graph  $(V, E \cup E_k^*)$  can be made  $k$ -connected by adding at most  $p((V, E) - S) - |E_k^*| - 1$  new edges.

**Step 1.** After setting  $H_1 := H$ ,  $F := E_H(s)$ , and  $E' := E(H_1) - F - E$ , we go to Step 2.

**Step 2.** Let  $G_1 := H_1 - s$ . If  $t(G_1) \leq \max\{2k - 3, k + 1\}$  holds, then halt outputting  $H^* := H_1$ .

Otherwise, while  $t(G_1) \geq \max\{2k - 2, k + 2\}$  holds, repeat the following procedure (A) or (B), where we execute procedure (A) if  $\beta(G_1) \leq \lceil t(G_1)/2 \rceil$  or procedure (B) otherwise.

**Procedure (A):**

- (A-1)** If there is a  $k$ -splittable pair  $\{(s, u_1), (s, u_2)\}$  such that  $H' = H_1 - \{(s, u_1), (s, u_2)\} + \{(u_1, u_2)\}$  satisfies  $t(H' - s) \leq t(G_1) - 1$ , then continue executing Step 2 after setting  $H_1 := H_1 - \{(s, u_1), (s, u_2)\} + \{(u_1, u_2)\}$ ,  $F := F - \{(s, u_1), (s, u_2)\}$ , and  $E' := E' \cup \{(u_1, u_2)\}$ .
- (A-2)** Otherwise, by Lemma 12,  $t(G_1) = 2k - 2$  holds and  $G_1$  can be made  $k$ -connected by adding at most  $2k - 4$  new edges. Output  $H^* := H_1$  as an solution satisfying (b).

*Property 6.* Each iteration of (A-1) decreases  $t(G_1)$  by at least one.  $\square$

**Procedure (B):** Choose a minimum disconnecting set  $S^*$  in  $G_1$  satisfying  $p(G_1 - S^*) = \beta(G_1)$ , where  $S^*$  is a shredder in  $G_1$  by  $\beta(G_1) \geq \lceil t(G_1)/2 \rceil + 1 \geq \max\{k, (k+2)/2 + 1\} \geq 3$ .

- (B-1)** If  $t(G_1) \geq \beta(G_1) + 1$  then, find a  $k$ -splittable pair  $\{(s, u_1), (s, u_2)\}$  in Lemma 10 so that  $H' = H_1 - \{(s, u_1), (s, u_2)\} + \{(u_1, u_2)\}$  with  $G' = H' - s$  satisfies  $\beta(G') \leq \beta(G_1) - 1$ . After setting  $H_1 := H'$ ,  $F := F - \{(s, u_1), (s, u_2)\}$ , and  $E' := E' \cup \{(u_1, u_2)\}$ , continue executing Step 2.
- (B-2)** Otherwise ( $\beta(G_1) = t(G_1)$ ), let  $S^*$  be a shredder in  $G_1$  with  $|S^*| = k - 1$  and  $p(G_1 - S^*) = \beta(G_1)$ . We distinguish the following three subcases in (B-2).
- (B-2-1)**  $G_1$  has an edge  $e = (v_1, v_2) \in E'$  with  $\{v_1, v_2\} - S^* \neq \emptyset$  and  $p(G_1 - e - S^*) = p(G_1 - S^*)$ . Then after hooking up the edge  $e$  at  $s$ , we split a pair  $\{(s, u_1), (s, u_2)\}$  in Lemma 11 such that the resulting graph  $H' := H_1 - \{(s, u_1), (s, u_2), (v_1, v_2)\} + \{(u_1, u_2), (s, v_1), (s, v_2)\}$  satisfies  $\kappa_{H'}(x, y) \geq k$  for all  $x, y \in V$  and  $G' = H' - s$  satisfies  $\kappa(G') = k - 1$ , and  $\beta(G' - S^*) = \beta(G_1 - S^*) - 1$ . After setting  $H_1 := H'$ ,  $F := F \cup \{(s, v_1), (s, v_2)\} - \{(s, u_1), (s, u_2)\}$ , and  $E' := E' \cup \{(u_1, u_2)\} - \{(v_1, v_2)\}$ , continue executing Step 2.
- (B-2-2)** There is an edge  $e = (v_1, v_2) \in E'$  with  $v_1, v_2 \in S^*$ . We replace the edge  $e$  with a new edge  $e' = (v_1, v_3)$  for a vertex  $v_3 \in V - S^*$ . Note that  $H' := H_1 - e + e'$  also satisfies  $\kappa_{H'}(x, y) \geq k$  for all  $x, y \in V$  by  $D \cap S^* = \emptyset$  for all  $D \in \mathcal{D}(G_1)$  (by Lemma 8) and we have  $V(e') - S^* \neq \emptyset$  and  $p((H' - s) - e - S^*) = p((H' - s) - S^*)$ . Then go to (B-2-1) after setting  $H_1 := H'$ , and  $E' := E' \cup \{e'\} - \{e\}$ .
- (B-2-3)** Every edge  $(u, v) \in E'$  satisfies  $u, v \in V - S^*$  and  $p((G_1 - \{e\}) - S^*) = p(G_1 - S^*) + 1$ . Output  $H^* := H_1$  as an solution satisfying (c).

*Property 7.* Each iteration of (B-1) and (B-2-1) decreases  $\beta(G_1)$  at least by one.  $\square$

Let us prove that a graph  $H^* = H_1$  in (B-2-3) satisfies condition (c). Since  $u, v \in V - S^*$  and  $p((G_1 - \{e\}) - S^*) = p(G_1 - S^*) + 1$  hold for all edges  $e = (u, v) \in E'$ , we have  $p((V, E) - S^*) = p(G_1 - S^*) + |E'|$ . Let  $\{T_1, \dots, T_p\} = \mathcal{C}(G_1 - S^*)$ , where  $p = p(G_1 - S^*)$ . For each  $T_i \in \mathcal{C}(G_1 - S^*)$ , let  $u_i \in \Gamma_{H_1}(s) \cap D_i$  with  $T_i \supseteq D_i \in \mathcal{D}(G_1)$ . We show that for a set  $E_1 = \{(u_i, u_{i+1}) \mid i = 1, \dots, p-1\}$  of new edges,  $G' := G_1 \cup E_1$  is  $k$ -connected. If  $G'$  has a disconnecting set  $S' \subset V$  with  $S' \neq S^*$  and  $|S'| = k - 1$ , then by Lemma 9(1)  $p(G' - S') = 2$  holds and there is an  $S'$ -component  $T' \subseteq T_i$  for some  $T_i \in \mathcal{C}(G_1 - S^*)$ , which contradicts that the edge  $(u_i, u_{i+1})$  or  $(u_{i-1}, u_i)$  connects  $T' (\supseteq D_i)$  and some  $T_j \in \mathcal{C}(G_1 - S^*) - \{T_i\}$  in  $G'$ . Therefore,  $G' = (V, E \cup E'')$  with  $E'' = E_k^* \cup E_1$  is  $k$ -connected, and  $|E_1| = p((V, E) - S) - |E_k^*| - 1$  holds.

We see that the algorithm  $\kappa\text{-SPLIT2}(H, s, F, k)$  runs in  $O(\min\{k, \sqrt{n}\} kn^2)$  time, as observed in the analysis of the complexity of  $\kappa\text{-SPLIT1}(H, s, k)$ .

## 9 Concluding Remarks

In this paper, we gave a polynomial time algorithm for augmenting a given  $\ell$ -connected graph  $G$  to a  $k$ -connected graph by adding at most  $\delta(k-1) + \max\{0, (\delta-1)(\ell-3)-1\}$  surplus edges over the optimum for  $k \geq 4$ ,  $\ell \geq 0$ , and  $\delta = k - \ell$ . However, in the case of  $\ell+1 = k \geq 4$ , Jordán's algorithm [10,11] produces at most  $(k-2)/2$  surplus edges over the optimum. Therefore, it is a future work to close the gap between this and our bound.

## References

1. J. Cheriyan and R. Thurimella, *Fast algorithms for  $k$ -shredders and  $k$ -node connectivity augmentation*, J. Algorithms, Vol.33, 1999, pp. 15–50.
2. K. P. Eswaran and R. E. Tarjan, *Augmentation problems*, SIAM J. Comput., Vol.5, 1976, pp. 653–665.
3. S. Even and R. E. Tarjan, *Network flow and testing graph connectivity*, SIAM J. Comput., Vol.4, 1975, pp. 507–518.
4. M. Grötschel, C. L. Monma and M. Stoer, *Design of survivable networks*, in: Handbook in Operations Research and Management Science, Vol.7, Network Models, North-Holland, Amsterdam, 1995, pp. 617–672.
5. T. Hsu, *Undirected vertex-connectivity structure and smallest four-vertex-connectivity augmentation*, Lecture Notes in Comput. Sci., 1004, Springer-Verlag, Algorithms and Computation (Proc. ISAAC '95), 1995, pp. 274–283.
6. T. Hsu and V. Ramachandran, *A linear time algorithm for triconnectivity augmentation*, Proc. 32nd IEEE Symp. Found. Comp. Sci., 1991, pp. 548–559.
7. T. Hsu and V. Ramachandran, *Finding a smallest augmentation to biconnect a graph*, SIAM J. Computing, Vol.22, 1993, pp. 889–912.
8. T. Ishii, *Studies on multigraph connectivity augmentation problems*, PhD thesis, Dept. of Applied Mathematics and Physics, Kyoto University, Kyoto, Japan, 2000.
9. T. Ishii, H. Nagamochi and T. Ibaraki, *Augmenting a  $(k-1)$ -vertex-connected multigraph to an  $\ell$ -edge-connected and  $k$ -vertex-connected multigraph*, Lecture Notes in Comput. Sci., 1643, Springer-Verlag, Algorithms (Proc. ESA '99), 1999, pp. 414–425.
10. T. Jordán, *On the optimal vertex-connectivity augmentation*, J. Combin. Theory Ser B, Vol.63, 1995, pp. 8–20.
11. T. Jordán, *A note on the vertex-connectivity augmentation problem*, J. Combin. Theory Ser B., Vol.71, 1997, pp. 294–301.
12. G. Kant, *Algorithms for drawing planar graphs*, PhD thesis, Dept. of Computer Science, Utrecht University, the Netherlands, 1993.
13. M. Kao, *Data security equals graph connectivity*, SIAM J. Discrete Math., Vol.9, 1996, pp. 87–100.
14. W. Mader, *Ecken vom Grad  $n$  in Minimalen  $n$ -fach zusammenhängenden Graphen*, Arch. Math. Vol.23, 1972, pp. 219–224.
15. H. Nagamochi and T. Ibaraki, *A linear-time algorithm for finding a sparse  $k$ -connected spanning subgraph of a  $k$ -connected graph*, Algorithmica, Vol.7, 1992, pp. 583–596.
16. H. Nagamochi and T. Ibaraki, *Computing edge-connectivity of multigraphs and capacitated graphs*, SIAM J. Discrete Math., Vol. 5, 1992, pp. 54–66.
17. T. Watanabe and A. Nakamura, *A minimum 3-connectivity augmentation of a graph*, J. Comput. System Sci., Vol.46, 1993, pp. 91–128.

# Locating Sources to Meet Flow Demands in Undirected Networks

Kouji Arata<sup>1</sup>, Satoru Iwata<sup>2</sup>, Kazuhisa Makino<sup>3</sup>, and Satoru Fujishige<sup>4</sup>

<sup>1</sup> Division of Systems Science, Graduate School of Engineering Science, Osaka University, Toyonaka, Osaka 560-8531 Japan. [arata@sflab.sys.es.osaka-u.ac.jp](mailto:arata@sflab.sys.es.osaka-u.ac.jp)

<sup>2</sup> Department of Mathematical Engineering and Information Physics, Graduate School of Engineering, University of Tokyo, Tokyo 113-8656 Japan  
[iwata@sr3.t.u-tokyo.ac.jp](mailto:iwata@sr3.t.u-tokyo.ac.jp)

<sup>3</sup> Division of Systems Science, Graduate School of Engineering Science, Osaka University, Toyonaka, Osaka 560-8531 Japan. [makino@sys.es.osaka-u.ac.jp](mailto:makino@sys.es.osaka-u.ac.jp)

<sup>4</sup> Division of Systems Science, Graduate School of Engineering Science, Osaka University, Toyonaka, Osaka 560-8531 Japan. [fujishig@sys.es.osaka-u.ac.jp](mailto:fujishig@sys.es.osaka-u.ac.jp)

**Abstract.** This paper deals with the problem of finding a minimum-cost vertex subset  $S$  in an undirected network such that for each vertex  $v$  we can send  $d(v)$  units of flow from  $S$  to  $v$ . Although this problem is NP-hard in general, Tamura et al. have presented a greedy algorithm for solving the special case with uniform costs on the vertices. We give a simpler proof on the validity of the greedy algorithm using linear programming duality and improve the running time bound from  $O(n^2M)$  to  $O(nM)$ , where  $n$  is the number of vertices in the network and  $M$  denotes the time for max-flow computation in the network with  $n$  vertices and  $m$  edges. We also present an  $O(n(m + n \log n))$  time algorithm for the special case with uniform demands and arbitrary costs.

## 1 Introduction

Let  $\mathcal{N} = (G, u, d, c)$  be an undirected network on the underlying graph  $G = (V, E)$  with the vertex set  $V$  and the edge set  $E$ . Let  $n = |V|$  and  $m = |E|$ . It is endowed with a capacity function  $u : E \rightarrow \mathbf{R}_+$ , a demand function  $d : V \rightarrow \mathbf{R}_+$ , and a cost function  $c : V \rightarrow \mathbf{R}_+$ , where  $\mathbf{R}_+$  denotes the set of nonnegative reals. This paper addresses the problem of finding a minimum-cost vertex subset  $S \subseteq V$  such that for each  $v \in V$  we can send  $d(v)$  units of flow from  $S$  to  $v$ .

For a pair of disjoint subsets  $X, Y \subseteq V$ , we denote by  $\lambda(X, Y)$  the maximum flow value between  $X$  and  $Y$  in  $\mathcal{N}$ . We simply write  $\lambda(v, Y)$  and  $\lambda(X, w)$  for  $v, w \in V$  instead of  $\lambda(\{v\}, Y)$  and  $\lambda(X, \{w\})$ , respectively. For convenience, we assign  $\lambda(X, Y) = +\infty$  if  $X \cap Y \neq \emptyset$ . Then the problem is formulated as follows.

$$\begin{aligned} & \text{Minimize} && \sum_{v \in S} c(v) \\ & \text{subject to} && S \subseteq V, \\ & && \lambda(S, v) \geq d(v) \quad (v \in V). \end{aligned} \tag{1}$$

We call this problem SOURCE LOCATION. We say that a vertex set  $S \subseteq V$  *covers* a vertex  $v$  if  $\lambda(S, v) \geq d(v)$ . Namely, SOURCE LOCATION asks for a minimum-cost subset  $S \subseteq V$  that covers all the vertices in  $V$ .

A special case of this problem with a constant cost function was introduced by Tamura et al. [11]. They called it *plural cover problem*. They first considered the case in which both  $d$  and  $c$  are constant and described an algorithm that runs in  $O(n^2M)$  time [10], where  $M$  denotes the time complexity for computing an  $s-t$  maximum flow in a given network  $\mathcal{N}$  [13, 14]. Later Tamura et al. [11] showed that a simple greedy algorithm solves problem SOURCE LOCATION in  $O(n^2M)$  time even if the demand function  $d$  is arbitrary while the cost function  $c$  is still constant. Ito et al. [5] described another algorithm to improve the time complexity to  $O(npM)$ , where  $p$  is the number of distinct values of  $d(v)$  ( $v \in V$ ), i.e.,  $p = |\{d(v) \mid v \in V\}|$ .

In this paper, we analyze the greedy algorithm of Tamura et al. [11] to give a simpler proof based on the linear programming duality. We then improve the greedy algorithm to run in  $O(nM)$  time.

As for the case in which the demand function  $d$  is constant, we give an  $O(n(m + n \log n))$  time algorithm. The algorithm makes use of maximum adjacency (MA) ordering (see Section 4 for MA ordering). The MA ordering has been used by Nagamochi and Ibaraki for solving the problems of minimum cut [6] and of edge-connectivity augmentation [7].

Finally, we show that SOURCE LOCATION is in general NP-hard. We show this by reducing the knapsack problem to SOURCE LOCATION. Hence, it remains open to prove the NP-hardness in the strong sense or to devise a pseudo-polynomial time algorithm.

We summarize the time complexity of SOURCE LOCATION in Table 1, where bold letters indicate the results obtained in this paper.

The rest of the paper is organized as follows. Section 2 formulates SOURCE LOCATION as an integer programming problem, Section 3 considers SOURCE LOCATION when the cost function  $c$  is constant, and Section 4 discusses the case in which the demand function  $d$  is constant. In Section 5, we show that SOURCE LOCATION is in general NP-hard.

## 2 Integer Programming Formulation

In this section, we formulate SOURCE LOCATION as an integer programming problem with an exponential number of constraints.

A *cut* is a proper nonempty subset of  $V$ . For a cut  $X$ , let  $\Delta X$  denote the set of edges that cross  $X$ , i.e.,  $\Delta X = \{e \mid e = (v, w) \in E, v \in X, w \in V - X\}$ , and  $\kappa(X)$  its capacity, i.e.,

$$\kappa(X) = \sum_{e \in \Delta X} u(e).$$

**Table 1.** Summary of the results on SOURCE LOCATION

	$c$ : constant	$c$ : arbitrary
$d$ : constant	$O(n^2M)$ $O(nM)$ $O(n(m + n \log n))$	Tamura et al. (1992) [10] Ito et al. (1997) [5]
		$O(n(m + n \log n))$
$d$ : arbitrary	$O(n^2M)$ $O(npM)$ $O(nM)$	Tamura et al. (1998) [11] Ito et al. (1997) [5]
		<b>NP-hard</b>

$M$ : the time complexity for computing a maximum  $s$ - $t$  flow in  $\mathcal{N}$ .

$p$ : the number of distinct values of  $d(v)$  ( $v \in V$ ).

If  $X = \{v\}$ , we write  $\kappa(v)$  instead of  $\kappa(\{v\})$ . For a disjoint pair of vertex subsets  $X, Y \subseteq V$ , we denote  $\kappa(X, Y) = \sum_{e \in \Delta X \cap \Delta Y} u(e)$ . For  $v \in V$ , we simply write  $\kappa(X, v)$  instead of  $\kappa(X, \{v\})$ .

We also denote by  $d(W)$  the maximum demand in  $W$ , i.e.,

$$d(W) = \max\{d(v) \mid v \in W\}.$$

We say that a vertex  $v \in W$  *attains the maximum demand in  $W$*  if  $d(v) = d(W)$ . A cut  $W$  is called *deficient* if  $\kappa(W) < d(W)$ . If a cut  $W$  is deficient and no other subset  $X \subset W$  is deficient,  $W$  is called a *minimal deficient set*.

**Lemma 1** ([11]). *Let  $\mathcal{N} = (G = (V, E), u, d, c)$  be an undirected network. Then  $S \subseteq V$  covers all vertices in  $V$  if and only if  $S \cap W \neq \emptyset$  holds for every minimal deficient set  $W$ .*

Let  $\mathcal{W} = \{W_1, W_2, \dots, W_l\}$  be the family of all the minimal deficient sets and let  $V = \{v_1, v_2, \dots, v_n\}$ . Define an  $l \times n$  matrix  $A = (A_{ij})$  by  $A_{ij} = 1$  if  $v_j \in W_i$  and  $A_{ij} = 0$  otherwise. From Lemma 1, SOURCE LOCATION can be written as the following 0-1 integer programming problem:

$$\begin{aligned} &\text{Minimize} && \sum_{j=1}^n c_j x_j \\ &\text{subject to} && \sum_{j=1}^n A_{ij} x_j \geq 1 \quad (i = 1, 2, \dots, l) \\ &&& x_j \in \{0, 1\} \quad (j = 1, 2, \dots, n), \end{aligned} \tag{2}$$

where  $c_j = c(v_j)$  ( $j = 1, 2, \dots, n$ ), and  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  is the characteristic vector of a subset of  $V$ .

*Remark 1.* Note that  $l = |\mathcal{W}|$  might be exponential in  $n = |V|$  and  $m = |E|$ . For example, let us consider a network  $\mathcal{N} = (G = (V, E), u, d, c)$ , where  $V = \{v_1, v_2, \dots, v_n\}$ ,  $E = \{(v_1, v_i) \mid i = 2, 3, \dots, n\}$ ,  $u(e) = 1$  ( $e \in E$ ) and

$$d(v_i) = \begin{cases} \lceil \frac{n}{2} \rceil & \text{if } i = 0 \\ 0 & \text{otherwise,} \end{cases}$$

and  $c$  is an arbitrary cost function. Then we can see that

$$\mathcal{W} = \{ W \mid |W| = \left\lfloor \frac{n}{2} \right\rfloor + 1, W \ni v_1 \},$$

and hence

$$|\mathcal{W}| = \binom{n-1}{\lfloor \frac{n}{2} \rfloor}.$$

□

### 3 The Uniform Cost Case

#### 3.1 A Greedy Algorithm

In this section, we consider SOURCE LOCATION with a constant cost function. Tamura et al. [11] proposed the following greedy algorithm to solve SOURCE LOCATION.

##### Algorithm GREEDY

**Step 0:** Arrange the vertices  $v_1, v_2, \dots, v_n$  in  $V$  such that  $d(v_1) \leq d(v_2) \leq \dots \leq d(v_n)$ .

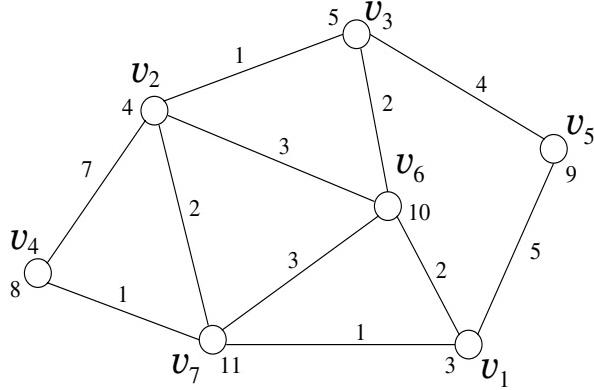
**Step 1:** Initialize  $j := 1$  and  $S := V$ .

**Step 2:** If  $S - \{v_j\}$  covers all vertices in  $V$ , then  $S := S - \{v_j\}$ .

**Step 3:** If  $j = n$  then output  $S$  and halt. Otherwise,  $j := j + 1$  and go to Step 2.

□

*Example 1.* Let us apply GREEDY to the network  $\mathcal{N} = (G = (V, E), u, d, c)$  given in Figure 1, where  $u$  and  $d$  are respectively attached to edges and vertices in Figure 1 and  $c(v) = 1$  for all  $v \in V$ . The results are illustrated in Figure 2. We initially include all vertices in  $S$  and check whether the set  $S - \{v_1\}$  covers all vertices or not. Since it covers  $v_1$  (and hence all vertices in  $V$ ), we update  $S := S - \{v_1\}$  (see (i)). We next check whether  $S - \{v_2\}$  covers all vertices or not. Since it covers both  $v_1$  and  $v_2$ , we update  $S := S - \{v_2\}$  (see (ii)). By repeating this argument to the current  $S = \{v_3, v_4, \dots, v_7\}$  (see (iii)–(vii)), we finally obtain  $S = \{v_4, v_5, v_7\}$  shown in (viii). □



**Fig. 1.** A network  $\mathcal{N} = (G = (V, E), u, d, c)$ , where  $c(v) = 1$  ( $v \in V$ ).

In order to show the correctness of algorithm GREEDY, we consider the linear programming relaxation of (2):

$$\begin{aligned} \text{Minimize} \quad & \sum_{j=1}^n c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^n A_{ij} x_j \geq 1 \quad (i = 1, 2, \dots, l), \\ & x_j \geq 0 \quad (j = 1, 2, \dots, n), \end{aligned} \quad (3)$$

and its dual:

$$\begin{aligned} \text{Maximize} \quad & \sum_{i=1}^l y_i \\ \text{subject to} \quad & \sum_{i=1}^l A_{ij} y_i \leq c_j \quad (j = 1, 2, \dots, n), \\ & y_i \geq 0 \quad (i = 1, 2, \dots, l). \end{aligned} \quad (4)$$

Recall that  $c_j = 1$  ( $j = 1, 2, \dots, n$ ) is assumed in this section.

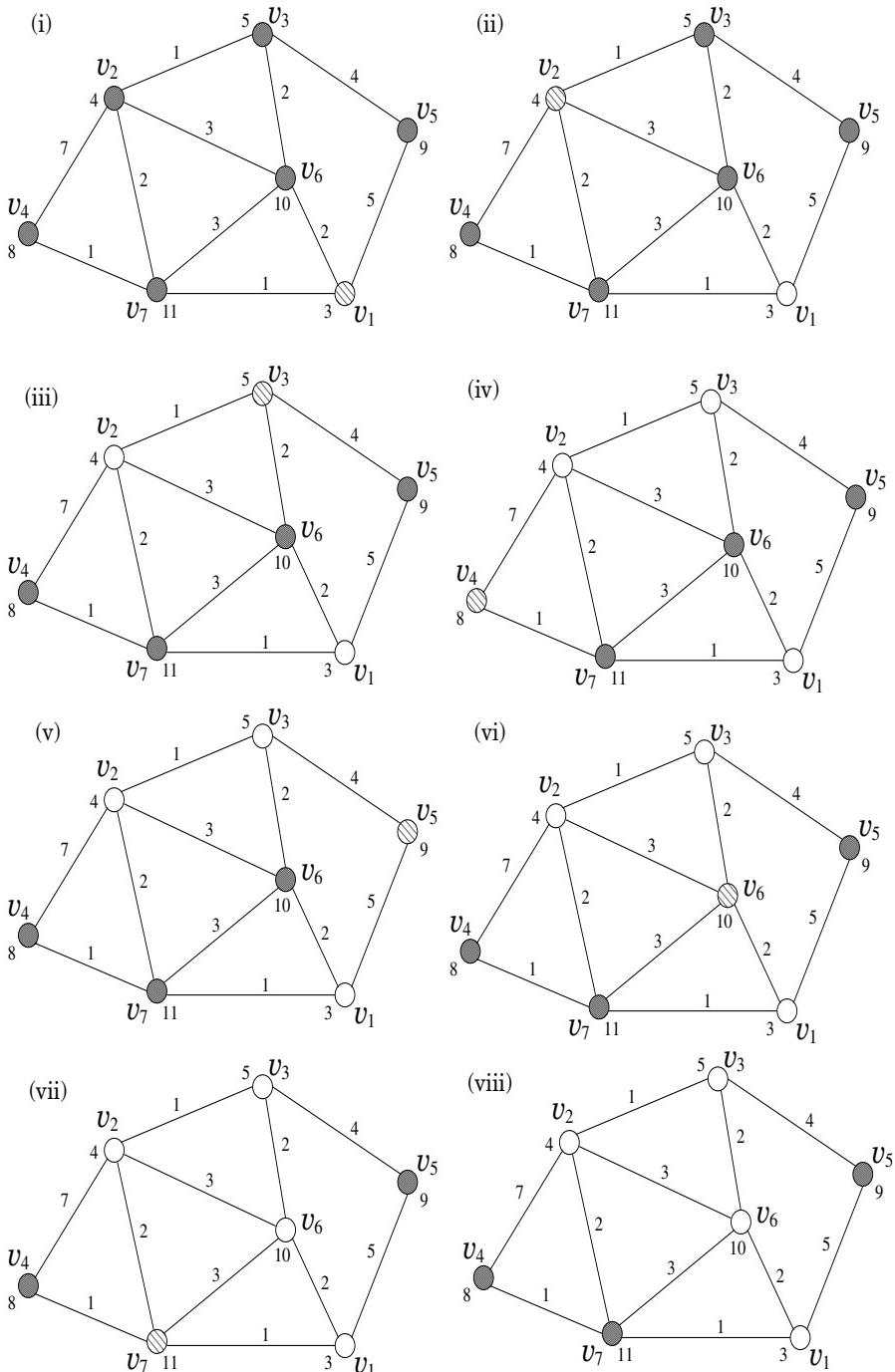
We also replace Steps 1 and 2 in algorithm GREEDY as follows.

**Step 1':** Initialize  $j := 1$ ,  $S := V$  and  $y_i := 0$  for  $i = 1, 2, \dots, l$ .

**Step 2':** (2-1) If  $S - \{v_j\}$  covers all vertices in  $V$ , then  $S := S - \{v_j\}$ .

(2-2) Otherwise, choose a  $W_i \in \mathcal{W}$  with  $W_i \cap S = \{v_j\}$ , and  $y_i := 1$ .

Note that Step 1' (initialization of  $y$ ) in the revised version might take exponential time (since  $|\mathcal{W}|$  might be exponential). However, this causes no trouble

Fig. 2. Algorithm GREEDY applied to the network  $\mathcal{N}$  in Figure \blacksquare

since we are now interested in the validity of the algorithm. Obviously, the algorithm always keeps a feasible solution  $S$  (i.e.,  $S$  covers all vertices in  $V$ ).

Let  $\mathbf{x}^*$  and  $\mathbf{y}^*$  be the primal and dual variables obtained at the end of the revised greedy algorithm. Note that  $\mathbf{x}^*$  is the characteristic vector of the output  $S$  of the algorithm.

The algorithm does not delete  $v_j$  from  $S$  if and only if it updates  $y_i$  as  $y_i := 1$  for some  $i$  with  $W_i \cap S = \{v_j\}$ . Hence, at the termination, we have

$$\sum_{j=1}^n x_j^* = \sum_{i=1}^l y_i^*. \quad (5)$$

Therefore,  $\mathbf{x}^*$  is a 0-1 solution satisfying (3) and (5). By the weak duality of linear programming problems (3) and (4), we only need to prove the feasibility of  $\mathbf{y}^*$  in (4) to show the correctness of the algorithm GREEDY. The feasibility of  $\mathbf{y}^*$  will be proved by Lemmas 2 and 3 given below.

Recall that the cut capacity function  $\kappa$  satisfies

$$\kappa(X) + \kappa(Y) \geq \kappa(X - Y) + \kappa(Y - X) \quad (X, Y \subseteq V). \quad (6)$$

A set function satisfying (6) is called *posi-modular* in [8].

**Lemma 2 ([11]).** *Let  $\mathcal{N} = (G = (V, E), u, d, c)$  be an undirected flow network. Let  $W_1$  and  $W_2$  be minimal deficient sets in  $\mathcal{N}$ , and for each  $i = 1, 2$ , let  $v_i \in W_i$  be a vertex that attain the maximum demand in  $W_i$ . If  $W_1 \cap W_2 \neq \emptyset$ , then we have  $v_1 \in W_1 \cap W_2$  or  $v_2 \in W_1 \cap W_2$ .*

*Proof.* Suppose, to the contrary, that both  $v_1 \in W_1 - W_2$  and  $v_2 \in W_2 - W_1$  hold. Since  $W_1$  and  $W_2$  are deficient sets,  $d(v_1) > \kappa(W_1)$  and  $d(v_2) > \kappa(W_2)$  hold. It follows from (6) that

$$\begin{aligned} d(v_1) + d(v_2) &> \kappa(W_1) + \kappa(W_2) \\ &\geq \kappa(W_1 - W_2) + \kappa(W_2 - W_1). \end{aligned}$$

This means that  $d(v_1) > \kappa(W_1 - W_2)$  or  $d(v_2) > \kappa(W_2 - W_1)$  holds. Since we have  $v_1 \in W_1 - W_2$  and  $v_2 \in W_2 - W_1$  by the assumption, it follows that  $W_1 - W_2$  or  $W_2 - W_1$  is deficient, which contradicts the minimality of  $W_1$  or  $W_2$ .  $\square$

Arrange the columns of  $A$  in such a way that  $d(v_1) \leq d(v_2) \leq \dots \leq d(v_n)$ . For each index  $i$  with  $1 \leq i \leq l$ , let  $k(i)$  denote the maximum number  $k$  with  $v_k \in W_i$ . Then Lemma 2 implies that the matrix  $A$  does not contain

$$i_1 \left( \begin{array}{ccc} j & k(i_1) & k(i_2) \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{array} \right) \quad (7)$$

as a submatrix.

**Lemma 3.** *The dual variable  $\mathbf{y}^*$  obtained by the revised greedy algorithm is feasible to (4).*

*Proof.* Suppose, to the contrary, that  $\mathbf{y}^*$  is infeasible. There is a pair of distinct rows,  $i_1, i_2$  and a column  $j$  such that  $y_{i_1}^* = y_{i_2}^* = 1$  and  $A_{i_1j} = A_{i_2j} = 1$ . Let  $j_0$  be the largest such number  $j$ , where we assume that the columns of  $A$  is already arranged in such a way that  $d(v_1) \leq d(v_2) \leq \dots \leq d(v_n)$ . Then we have  $k(i_1) \neq k(i_2)$  since otherwise  $y_{i_1}^*$  and  $y_{i_2}^*$  must be updated in the same iteration in Step 2', a contradiction. Note that  $j_0 = k(i_1)$  implies  $y_{i_2}^* = 0$  by the greedy algorithm. Hence we have  $j_0 < k(i_1)$ . Similarly, we also have  $j_0 < k(i_2)$ . Furthermore, we have  $A_{i_1k(i_2)} = 0$  due to the definition of  $j_0$ . Similarly, we have  $A_{i_2k(i_1)} = 0$ . This implies that  $A$  contains submatrix (7) forbidden by Lemma 2.  $\square$

We have thus shown the following.

**Theorem 1.** *If the cost function  $c$  is constant, then algorithm GREEDY produces an optimal solution of SOURCE LOCATION.*

### 3.2 An Efficient Implementation

We now analyze the time complexity of algorithm GREEDY. Steps 0, 1 and 3 are clearly executed in  $O(n \log n)$ ,  $O(1)$  and  $O(n)$  time, respectively. As for Step 2, Tamura et al. [11] checked if  $S - \{v_j\}$  covers all vertices in  $V$  by computing  $\lambda(S - \{v_j\}, v_i)$  (i.e., a max flow from  $S - \{v_j\}$  to  $v_i$ ) for all  $v_i$ . Clearly, this requires  $O(nM)$  time, where  $M$  is the time complexity for computing a maximum  $s$ - $t$  flow in the network  $\mathcal{N}$  [13,4]. Since Step 2 is iterated  $n$  times, the required time is  $O(n^2M)$  in total [11].

However, the following lemma implies that Step 2 can be replaced by

**Step 2'':** If  $S - \{v_j\}$  covers  $v_j$ , then  $S := S - \{v_j\}$ .

**Lemma 4.** *If  $S - \{v_j\}$  covers  $v_j$  in Step 2 of algorithm GREEDY, then  $S - \{v_j\}$  covers  $v_i$  for all  $i \leq j$ .*

*Proof.* Assume that some  $v_i$  with  $i < j$  is not covered by  $S - \{v_j\}$ . Then there exists a cut  $X$  with  $v_i \in X$ ,  $V - X \supseteq S - \{v_j\}$ , and  $\kappa(X) < d(v_i)$ . Then,  $S \cap X \subseteq \{v_j\}$  clearly holds. Moreover, we have  $S \cap X = \{v_j\}$  since otherwise  $S$  does not cover  $v_i$ , which contradicts the property that GREEDY always keeps a feasible set  $S$ . Hence,  $X$  separates  $v_j$  and  $S - \{v_j\}$ . Since  $\kappa(X) < d(v_i) \leq d(v_j)$ , it follows that  $S - \{v_j\}$  does not cover  $v_j$ , a contradiction.  $\square$

Thus we have improved the time complexity.

**Theorem 2.** *If the cost function  $c$  is constant, then problem SOURCE LOCATION can be solved in  $O(nM)$  time.*

## 4 The Uniform Demand Case

In this section, we consider SOURCE LOCATION with a constant demand function  $d$ . We assume that  $d(v) = g$  (a fixed positive real) holds for all  $v \in V$ . We show that it can be solved in  $O(n(m + n \log n))$  time without maximum flow computation. A key tool of the algorithm is the maximum adjacency (MA) ordering.

An ordering  $v_1, v_2, \dots, v_n$  of all vertices in  $V$  is called a *maximum adjacency (MA) ordering* if it satisfies

$$\kappa(\{v_1, v_2, \dots, v_i\}, v_{i+1}) \geq \kappa(\{v_1, v_2, \dots, v_i\}, v_j) \quad \text{for } 1 \leq i < j \leq n.$$

The MA ordering plays a crucial role in this section through the following lemma.

**Lemma 5 ([6,9]).** *Let  $G = (V, E)$  be an undirected graph with a nonnegative capacity function  $u$ . Then, the following statements hold.*

- (i) *An MA ordering  $v_1, v_2, \dots, v_n$  can be computed in  $O(m + n \log n)$  time.*
- (ii) *The last two vertices  $v_{n-1}$  and  $v_n$  for every MA ordering in  $G$  satisfy*

$$\lambda(v_{n-1}, v_n) = \kappa(v_n). \quad (8)$$

□

We mention here that we can choose the first vertex  $v_1$  arbitrarily.

Let us note that, if the demand function  $d$  is constant, minimal deficient sets are pairwise disjoint by the pos-modularity (6) of  $\kappa$ , i.e.,

$$W_1 \cap W_2 = \emptyset$$

holds for every pair of  $W_1$  and  $W_2$  in  $\mathcal{W}$ . Therefore, in order to solve SOURCE LOCATION, we try to find all minimal deficient sets  $W \in \mathcal{W}$  and construct a minimum-cost source set  $S \subseteq V$  by choosing from each  $W \in \mathcal{W}$  a vertex  $v \in W$  with the minimum cost  $c(v)$  among  $W$ .

Since any source set  $S$  must contain  $v \in V$  such that  $\kappa(v) < g$ , we initialize  $S$  as  $S := \{v \in V \mid \kappa(v) < g\}$ . To make use of MA orderings, we attach a new vertex  $s (\notin V)$  to a given network  $\mathcal{N}$  and, for each vertex  $v \in S$ , add the edge  $(s, v)$  with the capacity  $u(s, v) = g$ . By this modification of  $\mathcal{N}$ , every vertex  $v \in V$  satisfies  $\kappa(v) \geq g$ , i.e., either  $\kappa(v) \geq g$  holds in the original network or  $v \in S$  (i.e., the (modified) network  $\mathcal{N}$  contains the edge  $(s, v)$  with  $u(s, v) = g$ ). We then apply to the network  $\mathcal{N}$  an MA ordering  $v_0 (= s), v_1, \dots, v_{n-1}, v_n$  starting from  $s$ . By Lemma 5, we have

$$\lambda(v_{n-1}, v_n) = \kappa(v_n) \geq g.$$

Namely, every cut  $X$  that separates  $v_{n-1}$  and  $v_n$  satisfies  $\kappa(X) \geq g$ . This means that every minimal deficient set  $W \in \mathcal{W}$  (in the original network) that separates  $v_{n-1}$  and  $v_n$  forms  $W = \{v_{n-1}\}$  or  $\{v_n\}$ , since by the modification of  $\mathcal{N}$ , such a

$W$  must contain a vertex  $v \in V$  such that  $\kappa(v) < g$  in the original network, and hence we have  $|W| = 1$ . Since we already checked whether a cut  $X$  of the type  $X = \{v\}$  ( $v \in V$ ) is deficient, we do not have to consider the cut  $X$  separating  $v_{n-1}$  and  $v_n$ . We thus merge the vertices  $v_{n-1}$  and  $v_n$  into a single vertex  $\hat{v}$ , and check if  $\hat{v}$  satisfies  $\kappa(\hat{v}) \geq g$ . Since  $\kappa(\hat{v}) < g$  implies that  $W = \{v_{n-1}, v_n\}$  is a minimal deficient set in the original network, if  $\kappa(\hat{v}) < g$ , we update the network  $\mathcal{N}$  by adding edge  $(s, \hat{v})$  with the capacity  $u(s, \hat{v}) = g$ , and update  $S$  by adding  $v_{n-1}$  if  $c(v_{n-1}) < c(v_n)$ ; otherwise,  $v_n$ .

Now we have  $\kappa(\hat{v}) \geq g$  for all vertices except for  $s$  in the resulting network  $\mathcal{N}$ . By repeating the above argument for  $\mathcal{N}$  (i.e., we apply MA ordering  $v_0 (= s), v_1, \dots, v_{h-1}, v_h$  to  $\mathcal{N}$ , merge the last two vertices  $v_{h-1}$  and  $v_h$ , and so on), we can compute a minimum-cost source set  $S$ . Formally it can be written as follows.

#### Algorithm CONTRACT

**Input:** A network  $\mathcal{N} = (G = (V, E), u, d, c)$ , where  $d(v) = g$  for all  $v$ .

**Output:** A minimum-cost vertex set  $S \subseteq V$  which covers all vertices in  $V$ .

**Step 0:** Initialize  $S := \emptyset$ ,  $V' := V \cup \{s\}$ ,  $E' := E$ , and  $\alpha(v) := v$  for all  $v \in V$ .

**Step 1:** For each vertex  $v \in V$  such that  $\kappa(v) < g$ , put  $E' := E' \cup \{(s, v)\}$ ,  $u(s, v) := g$ , and  $S := S \cup \{\alpha(v)\}$ .

**Step 2:**

(2-I) Compute an MA ordering  $v_0 (= s), v_1, \dots, v_{h-1}, v_h$  starting from  $s$  in  $G' = (V', E')$ .

(2-II) Merge the last two vertices  $v_{h-1}$  and  $v_h$  in  $G'$  into a single vertex  $\hat{v}$ . Denote the resulting graph by  $G'$  again.

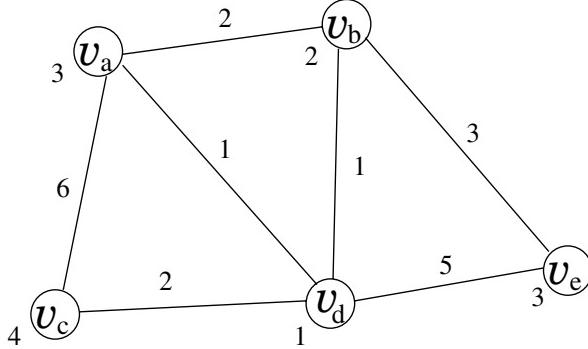
(2-III) If  $c(\alpha(v_{h-1})) < c(\alpha(v_h))$ , then  $\alpha(\hat{v}) := \alpha(v_{h-1})$ ; Otherwise,  $\alpha(\hat{v}) := \alpha(v_h)$ .

(2-IV) If  $\kappa(\hat{v}) < g$  in the current  $G'$ , then update  $E' := E' \cup \{(s, \hat{v})\}$ ,  $u(s, \hat{v}) := g$ , and  $S := S \cup \{\alpha(\hat{v})\}$ .

**Step 3:** If  $|V'| \leq 2$  or  $E'$  contains the edges  $(s, v)$  for all  $v \in V' - \{s\}$ , then output  $S$  and halt. Otherwise go to Step 2.  $\square$

Note that the algorithm prepares  $\alpha(\cdot)$  for computing from each  $W \in \mathcal{W}$  a vertex  $v \in W$  with the minimum cost  $c(v)$  among  $W$ . Formally,  $\alpha(v)$  ( $v \in V'$ ) stores the vertex  $v^*$  in the original network  $\mathcal{N}$  having the minimum cost  $c(v^*)$  among  $P_v$ , where  $P_v$  is the set of all vertices  $v$  in  $V$  which are merged to  $v$ .

*Example 2.* Let us apply Algorithm CONTRACT to the network  $\mathcal{N} = (G = (V, E), u, d, c)$  given in Figure 3, where  $u$  and  $c$  are respectively attached to edges and vertices in Figure 3 and  $d(v) = 8$  for all  $v$ . The results are illustrated in Figure 4. In Step 0, the algorithm initializes  $S$ ,  $V'$ ,  $E'$ , and  $\alpha$  (see (i)), and since  $v_8$  only satisfies  $\kappa(v_b) < 8$ , Step 1 updates the network (i) to (ii), and  $S := \{v_b\}$ . Step 2 then compute an MA ordering  $v_0 (= s), v_1 (= v_b), v_2 (= v_c), v_3 (= v_d), v_4 (= v_a), v_5 (= v_c)$  (see (iii-1)), and merge  $v_4 (= v_a)$  and  $v_5 (= v_c)$  into  $v_{ac}$  (see (iii-2)). Since  $c(\alpha(v_a)) < c(\alpha(v_c))$ , Step 2 puts  $\alpha(v_{ac}) := \alpha(v_a)$ . Moreover, by  $\kappa(v_{ac}) < 8$ , Step 2 updates  $E' := E' \cup \{(s, v_{ac})\}$ ,  $u(s, v_{ac}) := 8$ , and  $S := S \cup \{\alpha(v_{ac}) (= v_a)\}$  (see (iii-3)).



**Fig. 3.** A network  $\mathcal{N} = (G = (V, E), u, d, c)$ , where  $d(v) = 8$  ( $v \in V$ ).

Since  $|V'| = 5$  and  $(s, v_d) \notin E'$  for example, the algorithm returns to Step 2. Step 2 again compute an MA ordering  $v_0 (= s), v_1 (= v_b), v_2 (= v_{ac}), v_3 (= v_d), v_4 (= v_e)$  (see (iv-1)), and merge  $v_3 (= v_d)$  and  $v_4 (= v_e)$  into  $v_{de}$  (see (iv-2)). Since  $c(\alpha(v_d)) < c(\alpha(v_e))$ , Step 2 puts  $\alpha(v_{de}) := \alpha(v_d)$ . Moreover, by  $\kappa(v_{de}) < 8$ , Step 2 updates  $E' := E' \cup \{(s, v_{de})\}$ ,  $u(s, v_{de}) := 8$ , and  $S := S \cup \{\alpha(v_{de}) (= v_d)\}$  (see (iv-3)). Now  $E'$  contains the edges  $(s, v)$  for all  $v \in V' - \{s\}$ . Step 3 outputs  $S = \{v_a, v_b, v_d\}$  whose cost is 6.  $\square$

**Theorem 3.** Problem SOURCE LOCATION can be solved in  $O(n(m + n \log n))$  time if the demand function  $d$  is constant.

*Proof.* Since the above discussion shows the correctness of algorithm CONTRACT, we only consider its time complexity. Clearly Steps 0, 1 and 3 take  $O(n)$  time. Step 2 can be executed in  $O(n(m + n \log n))$  time since it has  $n - 1$  iterations and each iteration takes  $O(m + n \log n)$  time from Lemma 5. Therefore, in total, it requires  $O(n(m + n \log n))$  time.

## 5 NP-hardness of General Case

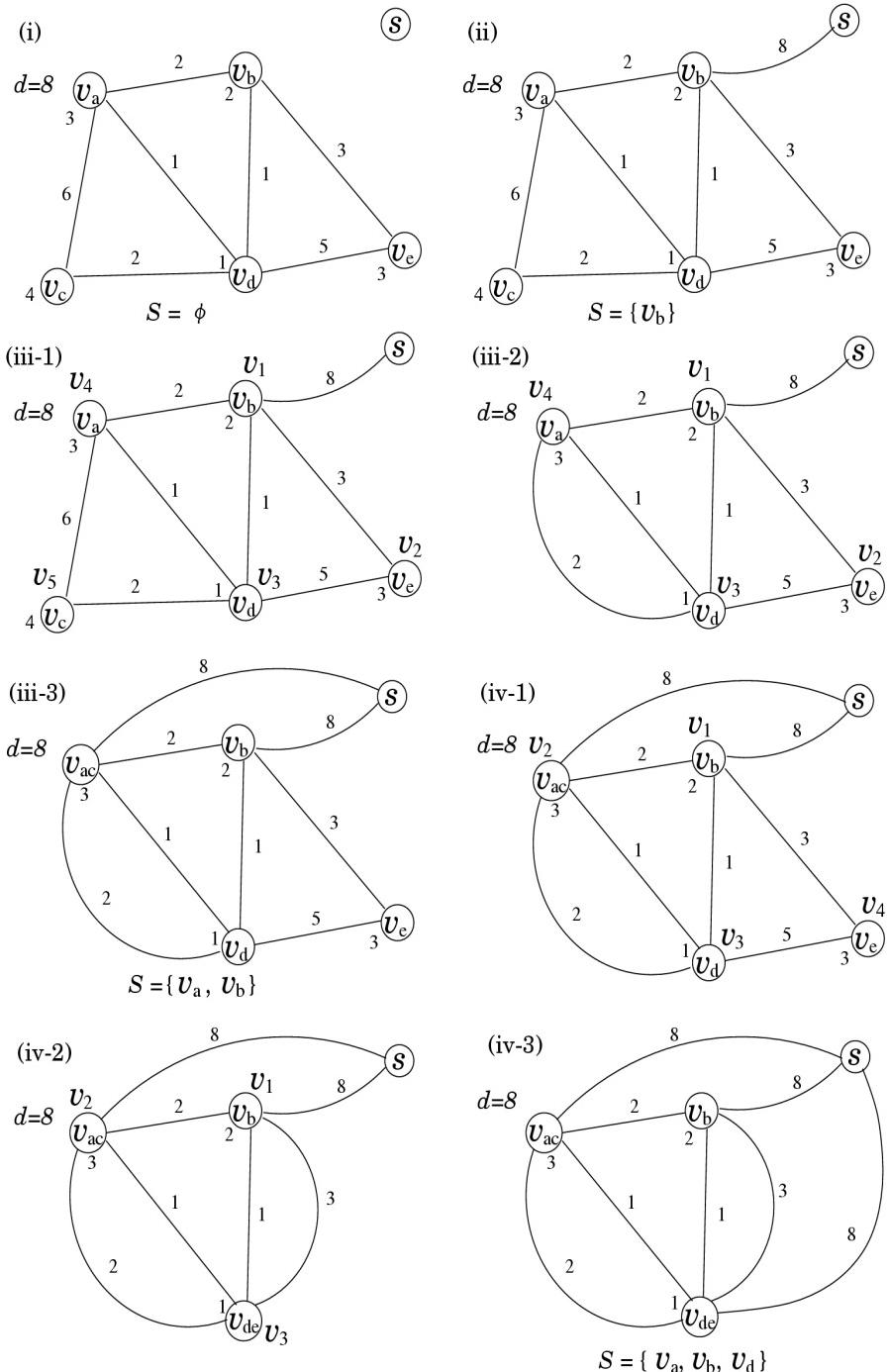
In this section, we show the NP-hardness of SOURCE LOCATION with non-constant cost and demand functions.

**Theorem 4.** Problem SOURCE LOCATION is NP-hard, even if the undirected graph  $G = (V, E)$  is a star, i.e.,  $E = \{(v, w) \mid w \in V \setminus \{v\}\}$  for some  $v \in V$ .

*Proof.* We transform Problem KNAPSACK to this problem, where KNAPSACK is known to be NP-hard [2].

### Problem KNAPSACK

**Input:** A finite set  $Z = \{z_1, z_2, \dots, z_n\}$  associated with a size function  $\sigma : Z \rightarrow \mathbf{Z}_+$  and a value function  $\omega : Z \rightarrow \mathbf{Z}_+$ , and positive integer  $b$  ( $\leq \sum_{z_i \in Z} \sigma(z_i)$ ). where  $\mathbf{Z}_+$  denotes the set of all nonnegative integers.



**Fig. 4.** Algorithm CONTRACT applied to the network  $\mathcal{N}$  in Figure 3.

**Output:** A subset  $X \subseteq Z$  that is an optimal solution of

$$\begin{aligned} & \text{Maximize} && \sum_{z \in X} \omega(z) \\ & \text{subject to} && \sum_{z \in X} \sigma(z) \leq b, \\ & && X \subseteq Z. \end{aligned} \tag{9}$$

It is easy to see that KNAPSACK is polynomially equivalent to the problem of computing a subset  $Y \subseteq Z$  that solves

$$\begin{aligned} & \text{Minimize} && \sum_{z \in Y} \omega(z) \\ & \text{subject to} && \sum_{z \in Y} \sigma(z) \geq \sum_{z \in Z} \sigma(z) - b, \\ & && Y \subseteq Z, \end{aligned} \tag{10}$$

by identifying  $Y$  with  $Z - X$ . Therefore, in the following we consider (10) instead of (9).

For this problem instance, we consider an undirected network  $\mathcal{N} = (G = (V, E), u, d, c)$  with  $V = Z \cup \{z_0\}$ ,  $E = \{(z_0, z_i) \mid z_i \in Z\}$ ,  $u(z_0, z_i) = \sigma(z_i)$  for  $i = 1, 2, \dots, n$  and

$$d(z_i) = \begin{cases} \sum_{z_i \in Z} \sigma(z_i) - b & \text{if } i = 0 \\ 0 & \text{otherwise,} \end{cases}$$

$$c(z_i) = \begin{cases} \sum_{z_i \in Z} \omega(z_i) + 1 & \text{if } i = 0 \\ \omega(z_i) & \text{otherwise.} \end{cases}$$

Note that  $d(z_i) = 0$  for all  $z_i \in Z$ . Therefore  $S \subseteq V$  covers all vertices in  $V$  if and only if it covers  $z_0$ , i.e.,

$$\lambda(S, z_0) \geq d(z_0) = \sum_{z_i \in Z} \sigma(z_i) - b. \tag{11}$$

Moreover, since  $\{z_0\}$  and  $Z$  covers  $z_0$ , and since  $c(z_0) > \sum_{z_i \in Z} c(z_i)$ , an optimal  $S$  is contained in  $Z$ . This implies that

$$\lambda(S, z_0) = \sum_{z_i \in S} u(z_0, z_i) = \sum_{z_i \in S} \sigma(z_i), \tag{12}$$

and hence (11) is equivalent to the constraint in (10).

Since  $c(z_i) = \omega(z_i)$  for all  $z_i \in Z$ ,  $S \subseteq Z$  is an optimal solution for the instance of problem (10) if and only if it is optimal for the corresponding instance for SOURCE LOCATION.  $\square$

## 6 Conclusion

In this paper, we have analyzed the greedy algorithm of Tamura et al. [11] for SOURCE LOCATION with a constant cost function and given a simpler proof based on the linear programming duality. We have also improved the greedy algorithm to run in  $O(nM)$  time. Moreover, we have given an  $O(n(m + n \log n))$  time algorithm for SOURCE LOCATION with a constant demand function. Finally, we have shown that SOURCE LOCATION is in general NP-hard by reducing KNAPSACK to SOURCE LOCATION.

## References

1. R. K. Ahuja, T. L. Magnanti and J. B. Orlin: *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, Englewood Cliffs, New Jersey, (1993).
2. M. R. Garey and D. S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freedman, New York, (1979).
3. A. V. Goldberg and S. Rao: Beyond the flow decomposition barrier, *Journal of the ACM*, **45** (1998), 783–797.
4. A. V. Goldberg and S. Rao: Flows in undirected unit capacity networks, *SIAM J. Discrete Mathematics*, **12** (1999), 1–5.
5. H. Ito, H. Uehara and M. Yokoyama: A faster and flexible algorithm for a location problem on undirected flow networks, *IEICE Trans. Fundamentals*, **E83-A**, 2000, to appear.
6. H. Nagamochi and T. Ibaraki: Computing edge-connectivity of multigraphs and capacitated graphs, *SIAM J. Discrete Mathematics*, **5** (1992), 54–66.
7. H. Nagamochi and T. Ibaraki: Deterministic  $\tilde{O}(nm)$  time edge-splitting in undirected graphs, *J. Combinatorial Optimization*, **1** (1997), 5–46.
8. H. Nagamochi and T. Ibaraki: A note on minimizing submodular functions, *Information Processing Letters*, **67** (1998), 169–178.
9. M. Stoer and F. Wagner: A simple min cut algorithm, *Journal of the ACM*, **44**, (1997) 585–591.
10. H. Tamura, M. Sengoku, S. Shinoda and T. Abe: Some covering problems in location theory on flow networks, *IEICE Trans. Fundamentals*, **E75-A** (1992), 678–683.
11. H. Tamura, H. Sugawara, M. Sengoku and S. Shinoda: Plural cover problem on undirected flow networks, *IEICE Trans. Fundamentals*, **J81-A** (1998), 863–869 (in Japanese).

# Improved Greedy Algorithms for Constructing Sparse Geometric Spanners

Joachim Gudmundsson<sup>1</sup>, Christos Levcopoulos<sup>1</sup>, and Giri Narasimhan<sup>2</sup> \*

<sup>1</sup> Department of Computer Science, Lund University, Box 118, 221 00 Lund, Sweden

[Joachim.Gudmundsson@cs.lth.se](mailto:Joachim.Gudmundsson@cs.lth.se), [Christos.Levcopoulos@cs.lth.se](mailto:Christos.Levcopoulos@cs.lth.se)

<sup>2</sup> Department of Mathematical Sciences, The University of Memphis,  
Memphis, TN 38152, USA

[Giri.Narasimhan@memphis.edu](mailto:Giri.Narasimhan@memphis.edu)

**Abstract.** Let  $G=(V,E)$  be a connected graph with positive weights and  $n$  vertices. A subgraph  $G'$  is a  $t$ -spanner if for all  $u,v \in V$ , the distance between  $u$  and  $v$  in the subgraph  $G'$  is at most  $t$  times the corresponding distance in  $G$ . We show a  $O(n \log n)$ -time algorithm which, given a set  $V$  of  $n$  points in  $d$ -dimensional space, and any constant  $t > 1$ , produces a  $t$ -spanner of the complete Euclidean graph of  $G$ . The produced spanner have  $O(n)$  edges, constant degree and weight  $O(\text{wt}(MST))$ .

## 1 Introduction

Spanners have applications in the design of geometric networks. Consider a set  $V$  of  $n$  points in  $R^d$ , where the dimension  $d$  is a constant. A geometric network on  $V$  can be modeled as an undirected graph  $G$  with vertex set  $V$  and with edges  $e = (u, v)$  of weight  $\text{wt}(e)$ . A Euclidean network is a geometric network where the weight of the edge  $e = (u, v)$  is equal to the Euclidean distance  $d(u, v)$  between its two endpoints  $u$  and  $v$ . For  $u, v \in V$ , let  $P$  be a  $uv$ -path in  $G$ , i.e., a path in  $G$  between  $u$  and  $v$ . The weight of the path  $P$  is denoted by  $\text{wt}(P)$  and is defined as the sum of the weights of the edges of  $P$ . Let  $t > 1$  be a real number. We say that  $G$  is a  **$t$ -spanner** for  $V$ , if for each pair of points  $u, v \in V$ , there exists a  $uv$ -path in  $G$  of weight at most  $t$  times the Euclidean distance between  $u$  and  $v$ . A **sparse  $t$ -spanner** is defined to be a  $t$ -spanner of size  $O(n)$  and weight  $O(\text{wt}(MST))$ . Given a geometric network  $G = (V, E)$ , a weight function  $w$  defined on its edges, and two vertices  $u, v \in V$ , we let  $D_{\{G,w\}}(u, v)$  denote the weight of the shortest path from  $u$  to  $v$  in  $G$  for the weight function  $w$ .

The problem of constructing spanners has been investigated by many researchers. Keil and Gutwin [5] showed that for any  $t > 1$ , and any set  $V$  of  $n$  points in the plane, a  $t$ -spanner for  $V$  having  $O(n)$  edges can be constructed in  $O(n \log n)$  time. Salowe [9], Vaidya [11] and Callahan and Kosaraju [2] showed the same result for any fixed dimension  $d$ . Das and Narasimhan [4] gave an  $O(n \log^2 n)$ -time algorithm that constructs for any set  $V$  of  $n$  points in  $R^d$ , and any constant  $t > 1$ , a  $t$ -spanner for  $V$  in which the degree of every point

---

\* Funded by NSF (CCR-940-9752) and Cadence Design Systems, Inc.

is bounded by a constant, and whose total edge weight is proportional to the weight of a minimum spanning tree of  $V$ . Chen et al. [3] showed that the lower bound for computing any  $t$ -spanner for a given set of points  $V$  in  $R^d$  is  $\Omega(n \log n)$  in the algebraic computation tree model.

Mount [6] has shown that a significant result claimed in Arya et al. [1] of an  $O(n \log n)$ -time algorithm to compute a sparse Euclidean spanner is incorrect. Thus the problem of devising an  $O(n \log n)$ -time algorithm to produce sparse spanners remained unsolved. Sparse spanners are also useful in designing efficient approximation schemes for geometric problems. In a startling development, Rao and Smith [8] showed an optimal  $O(n \log n)$ -time approximation scheme for the well-known Euclidean *traveling salesperson problem*, assuming that it is possible to compute sparse spanners in time  $O(n \log n)$ . Since the claim by Arya et al. [1] was incorrect, the existence of an  $O(n \log n)$ -time algorithm to construct sparse spanners has become a critical open problem. Note that the most efficient algorithm to construct sparse spanners is due to Das and Narasimhan [4] and runs in  $O(n \log^2 n)$  time. In this paper we design an algorithm that produces a  $t$ -spanner in time  $O(n \log n)$ , in the standard *real RAM model* defined in [7].

**Theorem 1.** *Given a set  $V$  of  $n$  points in  $d$ -dimensional space, and any real constant  $t > 1$ , a  $t$ -spanner of the complete Euclidean graph can be constructed in  $O(n \log n)$  time such that the spanner has  $O(n)$  edges, constant degree and weight  $O(\text{wt}(MST))$ . The constants in the  $O$ -notation depend on  $t$  and  $d$ .*

It was shown in [2] that the greedy algorithm produces spanners with  $O(n)$  edges and weight  $O(1) \cdot \text{wt}(MST)$ . However, a naive implementation of the greedy algorithm had a running time of  $O(n^3 \log n)$ , mainly due to the fact that  $\Omega(n)$  shortest path queries needed to be answered in a “dynamic” graph with at most  $O(n)$  edges, each of which could take  $O(n \log n)$  time.

Our algorithm is inspired by the algorithm due to Das and Narasimhan [4]. They showed how to use “clustering” in order to speed up shortest path queries. However, their algorithm was not efficient enough because they were unable to “maintain” the clusters efficiently and the algorithm had to frequently rebuild the clusters. For convenience, we will refer to the  $O(n \log^2 n)$ -time algorithm from [4] as the **DN-Clustering** spanner algorithm. We retain the general framework of that algorithm. Our main contribution is in developing techniques to efficiently perform “clustering”. We believe that the techniques that we have developed are likely to be useful in designing other greedy-style “dynamic algorithms”, i.e., in situations where only insertions take place and particularly in “increasing” order of length. What we prove in this paper is that after some preprocessing (which takes  $O(n \log n)$  time), given a linear-sized edge-weighted graph with integral edge weights in the range  $[0..N]$ , and given a set of “cluster-centers”, then we can perform “clustering” very efficiently in only  $O(n + N)$  time. In 1999, Thorup [10] showed that single source shortest path queries could be answered in linear time for undirected graphs with integer weights. The main reasons why this algorithm is not used in this paper is that it does not visit the vertices in order of increasing distance, which is crucial for our algorithm. Also,

it uses bit-shift for computing the floor function in constant time, which we do not allow in the real RAM model.

## 2 The DN-Clustering Spanner Algorithm

We first describe the cluster-based spanner algorithm by Das and Narasimhan [4]. It can be roughly described as follows. The algorithm starts with an empty spanner  $G'$ . A preprocessing step helps to eliminate all but a linear number of edges. Among the edges not eliminated, very short edges (i.e., those of length at most  $D/n$ , where  $D$  is the distance between the farthest pair of points) are simply added to  $G'$  since their contribution to the overall weight of the spanner cannot be more than the weight of a minimum spanning tree,  $\text{wt}(MST)$ . For the remaining edges, the greedy algorithm is simulated by sorting the edges, by increasing weight, and then processing them in  $\log n$  phases. Greedy processing of an edge  $e = (u, v)$  entails a shortest path query, i.e., checking whether  $D_{\{G', \text{wt}\}}(u, v) \leq t \cdot \text{wt}(e)$ . If the answer to the query is no, then  $e$  is added to the graph  $G'$ , else it is discarded. Whenever shortest path queries are required to be answered, these are not solved on the graph  $G'$  being constructed. Instead, they are solved on a “cluster-graph”  $H$ , which is simultaneously maintained. The cluster graph  $H$  from [4] has the following properties:

1. distances in  $H$  “closely” approximate distances in the current graph  $G'$ ,
2. every vertex in  $H$  has bounded degree, and
3. “specialized” shortest path queries in  $H$  can be answered in  $O(1)$  time.

The shortest path query when processing edge  $e = (u, v)$  is “specialized” in the sense that, at the instant that this query is processed, the cluster-graph  $H$  only has edges whose lengths are within a constant factor of  $\text{wt}(e)$ . For all practical purposes, cluster-graph  $H$  behaves like an unweighted graph of bounded degree for which a bounded radius subgraph around vertex  $u$  needs to be searched for the presence of vertex  $v$ . Since the edges considered have weights in the range  $(D/n, D]$  and they are processed in  $\log n$  phases, the edges can be sorted into  $\log n$  bins, where the  $i$ -th bin has edges of weight in the range  $(2^{i-1} \cdot D/n, 2^i \cdot D/n]$ . In order for shortest path queries to be answered quickly, the cluster-graph has to be carefully maintained. At the end of each phase, the cluster-graph is recomputed from scratch using the graph  $G'$ . This was deemed necessary since, in order to answer specialized shortest path queries about edge  $e = (u, v)$  in  $O(1)$  time, all edges in  $H$  need to be of length within a constant factor of  $d(u, v)$ .

The time complexity analysis is straightforward. Preprocessing steps ran in  $O(n \log n)$  time. The  $O(n)$  shortest path queries were processed in  $O(n)$  time, since each query took only  $O(1)$  time. The cluster graph computation at the start of each phase took  $O(n \log n)$  time (using Dijkstra’s algorithm on linear-sized graphs). Since there were  $\log n$  phases, the cluster-graph computations took a total of  $O(n \log^2 n)$  time. The crucial observation made in [4] was that shortest path queries need not be answered precisely. Instead, approximate shortest path queries suffice to produce low-weight spanners. The second observation was that shortest path queries are expensive if the shortest path involves a number of

small length edges, and that clustering can help eliminate all small length edges. This, of course, meant that the greedy algorithm, too, was only approximately simulated by the algorithm.

## 2.1 A Faster Spanner Algorithm

In this section, we present a simple modification to the DN-clustering algorithm to construct sparse  $t$ -spanners. This algorithm improves on the time complexity of the DN-clustering algorithm and runs in time  $O(\frac{n \log^2 n}{\log \log n})$  in the algebraic decision tree model of computation.

First we make the observation that there is wide disparity in the overall time spent by the DN-clustering algorithm on shortest path queries ( $O(n)$ ) and the time spent on the cluster-graph computations ( $O(n \log^2 n)$ ). In order to balance out the two costs, it is necessary to do fewer than  $O(\log n)$  cluster-graph computations, which in turn would make the shortest path queries more expensive. Instead of processing the edges in  $\log n$  phases, we process them in  $\frac{4 \cdot d \cdot \log n}{\log \log n}$  “batches”. We use the term batches to distinguish from the word phases used by the earlier DN-clustering algorithm. If “clustering” is recomputed after processing every batch of edges, since each call to the clustering algorithm takes  $O(n \log n)$  time, the total time for cluster graph computations will be  $O(\frac{n \log^2 n}{\log \log n})$ . We carefully analyze the cost of the  $O(n)$  shortest path queries and show that it can now be answered in a total of  $O(n \log n)$  time. In [4], in phase  $i$ , edges from the  $i$ -th bin were processed. These edges had weights in the range  $(W, 2W]$ , where  $W = 2^{i-1}(D/n)$ . During phase  $i$ , the cluster graph  $H$  could have edges (“inter-cluster” edges) whose weights were in the range  $(\delta W, 2W(1+2\delta)]$ . This meant that for edge  $e = (u, v)$  of weight  $l \in (W, 2W]$ , checking whether there is path from  $u$  to  $v$  of length at most  $t \cdot l$  could be done in  $O(1)$  time. More precisely, it was observed in [4] that if there does exist a path from  $u$  to  $v$  of length at most  $t \cdot l$ , then the number of edges on this path can be at most  $\frac{2t}{\delta}$  (since  $l \leq 2W$ ). It was further observed that since the vertices of  $H$  had a constant degree bound (say  $c$ ), and since there are at most  $O(c^{\frac{2t}{\delta}})$  vertices that lie  $\frac{2t}{\delta}$  edges away from vertex  $u$ , this shortest path query could be done in  $O(c^{\frac{2t}{\delta}} \log c^{\frac{2t}{\delta}})$  time (running Dijkstra’s algorithm starting from vertex  $u$  suffices). A tighter analysis was unnecessary in the DN-Clustering algorithm of [4] since  $c$ ,  $t$ , and  $\delta$  were all constants); below we show an improved analysis of this cost.

Recall that our algorithm works in  $\frac{4 \cdot d \cdot \log n}{\log \log n}$  batches. Batch  $i$  of our algorithm can be described as follows. For  $W = 2^{\frac{(i-1) \cdot \log \log n}{4 \cdot d}}(D/n)$ , the edges processed in batch  $i$  have weights in the range  $(W, W2^{\frac{\log \log n}{4 \cdot d}}]$ , i.e., they are in the range  $(W, W(\log n)^{\frac{1}{4 \cdot d}}]$ . This implies that, for edge  $e = (u, v)$  of weight  $l \in (W, W(\log n)^{\frac{1}{4 \cdot d}}]$ , we need to check whether there is a path from  $u$  to  $v$  of length at most  $t \cdot l$ . During batch  $i$ , the cluster graph  $H$  could have edges (“inter-cluster” edges) whose weights are in the range  $(\delta W, (1+2\delta)W(\log n)^{\frac{1}{4 \cdot d}}]$ . Thus, if there does exist such a path from  $u$  to  $v$ , then the number of edges on this path can be at most  $\frac{t(\log n)^{\frac{1}{4 \cdot d}}}{\delta}$ . The crucial observation we make is that the

vertices of the cluster-graph correspond to clusters of radius  $\delta W$ . These clusters may overlap, but their centers can lie in only one cluster. In other words, if these clusters are shrunk in half, they do not intersect. Thus the vertices correspond to disjoint clusters of radius  $\delta \cdot W/2$ . Now, it is possible to bound the number of vertices within distance at most  $t \cdot l = tW(\log n)^{\frac{1}{4-d}}$ . A simple packing argument shows that the number of balls of radius  $r$  that can be packed in a ball of radius  $R$  is bounded by  $O((R/r)^d)$ , where  $d$  is the dimension of the space. In our case, the number of balls of radius  $r = \frac{\delta W}{2}$  that can be packed in a ball of radius  $R = tW(\log n)^{\frac{1}{4-d}}$  is at most  $O((\frac{t \cdot (\log n)^{\frac{1}{4-d}}}{\delta^2})^d)$ . Thus the maximum number of vertices (and edges, due to the constant degree) that can be reached when performing Dijkstra's algorithm starting from vertex  $u$  is  $O((\frac{t \cdot (\log n)^{\frac{1}{4-d}}}{\delta^2})^d)$ . Since  $t$ ,  $d$  and  $\delta$  are constants,  $O((\frac{t \cdot (\log n)^{\frac{1}{4-d}}}{\delta^2})^d) = O((\log n)^{\frac{1}{4}})$ . We conclude that Dijkstra's algorithm for a shortest path query has a time complexity of  $O((\log n)^{\frac{1}{4}} \cdot (\log((\log n)^{1/4}))) = O(\log n)$ .

The obvious consequence is that all  $O(n)$  shortest path queries can be answered in  $O(n \log n)$  time, and hence, we have proved the following theorem:

**Theorem 2.** *In the algebraic decision tree model of computation, given a set  $V$  of  $n$  points in  $d$ -dimensional space, and any real constant  $t > 1$ , a  $t$ -spanner of the complete Euclidean graph can be constructed in  $O(\frac{n \log^2 n}{\log \log n})$  time such that the spanner has  $O(n)$  edges, constant degree and weight  $O(1) \cdot \text{wt}(MST)$ . The constants implicit in the  $O$ -notation depend on  $t$  and  $d$ .*

### 3 An Improved Spanner Algorithm

In the rest of the paper, we describe an efficient algorithm to construct sparse spanners with a running time of  $O(n \log n)$ . The running time of  $O(n \log n)$  for our algorithm is achieved by designing an  $O(n)$ -time algorithm for the clustering step, thus executing all the clustering steps in  $O(n \log n)$  time. Note that the running time is  $O(n \log n)$  even if clustering is executed  $O(\log n)$  times.

One crucial idea that we employ to speed up the clustering is to replace the real-valued weights by integral values. As observed in [4], the shortest path queries required by the algorithm need not be answered precisely; approximately correct answers suffice. A convenient way to achieve the *integralization* is to use the *floor/ceiling* function. However, this assumes a more powerful model of computation. In order to get around this problem, we compute the  $O(n)$  floor/ceiling functions needed by using operations allowed under the RAM model. The second crucial component of our algorithm is an implementation of the clustering algorithm in  $O(n)$  time assuming small integral weights for the edges. We also prove that the integralization introduces only a bounded amount of error, and that this error retains the correctness of the other required operations.

The improved spanner algorithm can be roughly described as follows. It is important to note that the skeleton of the algorithm is similar to the DN-clustering

algorithm from [4]. In particular, this improved algorithm also runs in  $O(\log n)$  phases. The algorithm starts with an empty graph  $G'$  and employs the same preprocessing step to eliminate all but a linear number of edges. This step is done by a call to the  $t$ -spanner algorithm presented by Arya et. al. in [1], with  $\sqrt{t/t'}$  as input parameter. Note that this algorithm in [1] is correct and runs in time  $O(n \log n)$ . It also guarantees that the graph has constant degree. As before, short edges of length at most  $D/n$  are simply added to  $G'$ ; their contribution to the overall weight of the spanner is bounded by  $wt(MST)$ . The greedy algorithm is now simulated on the remaining edges and the edges are added to the graph  $G'$ . The edges of the graph have real-valued weights that are equal to the Euclidean distance between their endpoints. The edges are sorted by increasing weight and then processed in  $\log n$  phases. Each of the edges also have corresponding integer-valued weights that are sufficiently close approximations of the real-valued weights; these integer-valued weights change through the course of the algorithm. In order to distinguish between the real- and integer-valued weights, we assume that there are two different weight functions defined on the edges of  $G'$ . For edge  $e = (u, v)$ , the real-valued weight function  $wt(e)$ , as mentioned before, is defined as the Euclidean distance  $d(u, v)$  between  $u$  and  $v$ . The integer-valued weight function denoted by  $Iwt_i(e)$  is a function of  $wt(e)$  and the phase number  $i$  is maintained by the algorithm as described later. Whenever the phase number is clear by the context, we use the simpler notation  $Iwt(e)$  instead of  $Iwt_i(e)$ . Also, unless specified otherwise, we assume that when we refer to the weight of an edge, we are referring to the real-valued weight of the edge. At the start of each phase, the integer-valued weight function  $Iwt(e)$  is recomputed for this phase. Then a set of vertices of  $G'$  are selected as cluster-centers and a cluster graph  $H$  is constructed from the current spanner graph  $G'$  (using the weight function  $Iwt$ ); this graph  $H$  is a simpler graph than the graph  $G'$  and distances between vertices in  $H$  are reasonably close to distances between the same pair of vertices in  $G'$ . The differences of this from the one in [4] lies in the fact that the cluster-centers have to be selected before the clustering is done and the clustering is done with the weight function  $Iwt$ . As mentioned before, we improve on the time complexity of this clustering step and show how it can be implemented to run in  $O(n)$  time. Once the cluster graph  $H$  is constructed, the algorithm processes the set of edges for that phase. Greedy processing of an edge  $e = (u, v)$  entails a shortest path query, i.e., checking whether  $D_{\{G', wt\}}(u, v) \leq t \cdot wt(e)$ . As in [4], this query is answered in  $O(1)$  time per query by performing an approximate shortest path query on the simpler graph  $H$ . If the answer to the query is yes, then edge  $e$  is added to the graph  $G'$ , else it is discarded. Each of the steps is described in more detail in the rest of the paper.

Since the edges that remain to be considered have weights in the range  $(D/n, D]$  and they are processed in  $\log n$  phases, the edges can be sorted into  $\log n$  bins, where the  $i$ -th bin has edges of weight in the range  $(2^{i-1} \cdot D/n, 2^i \cdot D/n]$ . At the start of each of the  $\log n$  phases, the algorithm calls the “clustering” algorithm, which is required to answer the shortest path queries efficiently. The clustering algorithm is described in section 3.2. Later we show that the running

time of our algorithm is  $O(n \log n)$ . Note that processing is done in  $\log n$  phases. If fewer number of phases are used as in the faster spanner algorithm described in section 2.1, then the error due to integralization could be too large. Even if fewer number of phases can be used, the running time of the overall algorithm will remain as  $O(n \log n)$ , since it is dominated by other steps in the algorithm. In particular, the integralization itself has an initial cost of  $O(n \log n)$ .

The detailed algorithm is given below in Fig. 1.

**Algorithm** IMPROVED-GREEDY( $V, t, t'$ )

1. Compute a  $(\sqrt{t/t'})$ -spanner  $G = (V, E)$  using the algorithm from [1]
2.  $\delta := \min \left( \frac{\sqrt{tt'} - t'}{4(\sqrt{tt'} + 3t')}, \frac{\sqrt{tt'} + 34\sqrt{tt'} + 1 - (\sqrt{tt'} + 5)}{24} \right)$
3. sort  $E$ ;  $D :=$  weight of largest edge in  $E$ ;
4.  $W_0 := 0$ ;  $W_i := 2^{(i-1)}D/n$  for  $i = 1, 2, \dots, \log n$
5.  $I_i := (W_i, W_{i+1}]$  for  $i = 0, 1, \dots, (\log n - 1)$
6.  $E_i :=$  (sorted) edges of  $E$  with weights in  $I_i$ ;  $E' := E_0$ ;  $G' := (V, E')$ ;
7. INTEGRALIZE( $E_0, 0$ )
8.  $C_1 :=$  NAIVE-CENTERS( $G', \delta \cdot W_1$ );  $M_1 := \emptyset$
9. **for**  $i := 1$  to  $\log n$  **do**
10.     INTEGRALIZE( $E_i, i$ )
11.     REINTEGRALIZE( $E_0 \cup E_1 \cup \dots \cup E_{i-1}$ )
12.      $H :=$  CLUSTER-GRAPH( $G', Iwt, C_i, r, R$ )
13.     **for** each edge  $e = (u, v) \in E_i$  **do**
14.         **if** not SHORT-PATH( $H, u, v, \sqrt{tt'} \cdot d(u, v)$ ) **then**
15.              $E' := E' \cup \{e\}$ ;  $G' := (V, E')$
16.      $C_{i+1} :=$  UPDATE-CENTERS( $H, i, C_i, r$ )
17. output  $G'$

**Fig. 1.** The  $O(n \log n)$ -time spanner algorithm

### 3.1 Integralization

As mentioned before, in order to speed up the cluster-graph computation, we replace the real-valued edge weights by integral values. The integralization changes in every phase. It is done in such a way that the edge weights and distances encountered in that phase are always in the range  $[0..N]$ , where  $N = c \cdot n$  for some constant integer  $c$ . The choice of  $c$  will dictate the errors introduced in the distance computations; this will be discussed later.

A closer inspection of a phase leads to the following simple observations. At the start of phase  $i$ , the spanner graph constructed so far has edges of weight at most  $W_i$ . During phase  $i$ , the edges considered for inclusion by the greedy algorithm are in the range  $(W_i, 2W_i]$ . The shortest path queries for an edge of length  $l$  involves checking whether the distance between a given pair of vertices

is at most  $t \cdot l$ . Hence the longest paths that need to be dealt with during phase  $i$  are of weight  $t \cdot 2W_i$ . The idea is to make the largest distance to correspond to the integer  $c \cdot n$ . To be on the safe side, since there are small errors in the distance computations, we set  $2t \cdot 2W_i$  to correspond to  $c \cdot n$ . Thus, in phase  $i$ , unit integer length will correspond to real length of  $U_i = \frac{4t \cdot W_i}{c \cdot n}$ .

Although a constant-time floor/ceiling function is not used in the algorithm, a convenient way to describe the integralization is as follows:  $Iwt_i(e) := \lceil \frac{wt(e)}{U_i} \rceil$ .

**Error Bounds:** Assuming the integralization defined above, we observe that the function  $Iwt$  always involves a “rounding up”. Hence,  $Iwt_i(e) \cdot U_i \geq wt(e)$ . It is also easy to see that in phase  $i$ , the error in the length of any single edge of the spanner graph is at most  $U_i$ . In other words,  $Iwt_i(e) \cdot U_i - wt(e) \leq U_i$ . Note that this error is an additive or an absolute error. Since any simple path can use at most  $n - 1$  edges, it is also easy to see that the error in the length of any simple path of the spanner graph is at most  $nU_i$ . Another consequence is that given two simple paths  $P_1$  and  $P_2$ , if  $Iwt(P_1) = Iwt(P_2)$ , then  $|wt(P_1) - wt(P_2)| \leq nU_i$ . It follows that  $nU_i$  is also a bound on the error that can be introduced when running Dijkstra’s single-source-shortest-path algorithm using the integral weights instead of the real weights. The following lemma formalizes this statement:

**Lemma 1.** *In phase  $i$ , given any  $\epsilon > 0$  and given vertices  $u$  and  $v$  in  $G'$  such that  $D_{\{G', wt\}}(u, v) \geq W_i$ ,  $D_{\{G', wt\}}(u, v) \leq D_{\{G', Iwt\}}(u, v) \cdot U_i < (1 + \epsilon) \cdot D_{\{G', wt\}}(u, v)$ .*

*Proof.* We give a sketch of the proof. In phase  $i$ , for a path  $P$  such that  $wt(P) \geq W_i$ , the error in computing its weight is at most  $nU_i$ . Thus the relative error (i.e., the ratio of the error to the weight of the path) is at most  $nU_i/W_i = \frac{4t}{c}$ . The proof follows by setting  $\epsilon = \frac{4t}{c\delta} > \frac{4t}{c}$  and using the well-known property of Dijkstra’s algorithm that the minimum value in the priority queue is monotonically non-decreasing. Note that  $\epsilon$  can be made as small as desired by choosing an appropriate value of  $c$ .  $\square$

**Corollary 1.** *For a path  $P$  in  $G'$  with  $wt(P) \geq \delta W_i$  (i.e.,  $Iwt(P) \geq R$ ), the absolute error in computing its weight is at most  $nU_i$ , and the relative error is at most  $\frac{nU_i}{\delta W_i} = \epsilon$ , for any  $\epsilon > 0$ .*

**Computing the Integralization.** Here we show how to compute the integer values of the weights of the edges over all phases in  $O(n \log n)$  time without using the floor/ceiling function.

We first observe that the spanner graph has at most  $O(n)$  edges at the start of any phase. Consider a specific phase  $i$ . In this phase, for a specific edge, since its integer value is in the range  $[0..N]$  (where  $N = c \cdot n$ ), it can be computed in  $O(\log n)$  time without the use of the floor/ceiling function by performing a binary search on the set of real values  $j \cdot U_i$ , for  $j = 0, \dots, N$ . We assume that

the function  $\text{INTEGRALIZE}(E, i)$  performs this operation for each edge in the set  $E$  in  $O(\log n)$  time per edge.

If the above observations are used in a naive fashion for all edges, then the cost of integralization is  $O(n \log n)$  just for one phase. Since the number of phases is not a constant, the integralization would turn out to be too expensive. Our algorithm spends  $O(\log n)$  time for computing the integralization of an edge weight over all the phases. The idea is to compute the integral value in  $O(\log n)$  time when the edge is encountered for the first time. Integralizations of an edge for subsequent phases is done by calling  $\text{REINTEGRALIZE}$ , and are computed in  $O(1)$  time from the integer weights of the edge computed in the previous phase. If the integral weight of an edge is  $I$  in phase  $i$ , then the integral weight of the edge in phase  $i+1$  will be  $I/2$  if  $I$  is even, and  $(I+1)/2$  if it is odd. This is correct since  $U_{i+1} = 2U_i$ , i.e., the integralization in phase  $i+1$  is twice as coarse as that in phase  $i$ . Checking if an integer is odd or even cannot be done in constant time in the real RAM model, but can quite easily be accomplished by using  $O(n \log n)$  preprocessing. One way to accomplish this would be to build a balanced binary tree including  $c \cdot n$  elements with the values  $1 \dots c \cdot n$ . Every element in the tree also contains a pointer to the element in the tree containing the value  $\lceil \frac{val}{2} \rceil$ . This value can be computed in time  $O(\log n)$  and searching the tree for the value is also done in  $O(\log n)$ . Hence, by using  $O(n \log n)$  time preprocessing, the integral weight of an edge for the next phase can be computed in constant time. Note also that the relative error for an edge with newly computed weight is less than  $U_{i+1}$ , hence Lemma 1 still holds. It is clear that  $\text{REINTEGRALIZE}(F)$  performs its operation for each edge in the edge set  $F$  in  $O(1)$  time per edge.

The above explanation proves that the integralization is computed in time  $O(n \log n)$  for all edges over all phases. The integer weights are then used directly in the clustering algorithms described below.

### 3.2 Clustering the Graph

First for some definitions. Here we assume that  $G = (V, E)$  is an arbitrary weighted graph, with weight function  $w$  defined on the edges in  $E$ . The following definition of a cluster is modified from the one in [4] to allow for arbitrary weight functions. The definition of a cluster-cover is also modified and is defined for a given set of cluster-centers.

#### Definition 1. Cluster, cluster-center, and radius

*Given a Euclidean graph  $G = (V, E)$ , a vertex  $v \in V$ , a radius  $r$ , and a weight function  $w$  defined on the edges in  $E$ ,  $\text{CLUSTER}(G, v, r, w)$  is defined as the set of all vertices  $u$  such that  $D_{G,w}(v, u) \leq r$ . The vertex  $v$  is called the cluster-center of this cluster and  $r$  is called the radius of the cluster.*

#### Definition 2. Cluster-cover

*Given a Euclidean graph  $G = (V, E)$ , a set  $C = \{v_1, v_2, \dots, v_m\} \subseteq V$ , a radius  $r$ , and a weight function  $w$  defined on the edges in  $E$ , the  $\text{CLUSTER-COVER}(G, C, r, w)$  is defined as a set of clusters  $\mathcal{K} = \{K_1, K_2, \dots, K_m\}$  such*

that  $K_i$  is a cluster with radius  $r$  and cluster-center at  $v_i$  ( $i = 1, 2, \dots, m$ ), and such that  $K_1 \cup K_2 \cup \dots \cup K_m = V$ .

During the course of the algorithm, clustering is performed on the spanner graph  $G'$  with the weight function  $Iwt$ . Also, the set  $C$  and the value  $r$  will be chosen in such a way that the cluster-cover always exists. In general, clusters in a cluster-cover may overlap. We also modify the definition of a cluster-graph so that it is a bit more general and it is defined for a given set of clusters and for an arbitrary weight function.

### Definition 3. Cluster graph

Assume that  $G = (V, E)$  is a Euclidean graph with a weight function  $w$  defined on its edges. Assume that  $C = \{v_1, v_2, \dots, v_m\} \subseteq V$  is a given set of cluster-centers. For a given radius  $r$ , we assume that  $\mathcal{K} = \{K_1, K_2, \dots, K_m\}$  is equal to  $\text{CLUSTER-COVER}(G, C, r, w)$ . Given  $R \geq r$ , the  $\text{CLUSTER-GRAFH}(G, w, C, r, R)$  is defined as a graph  $H = (V, E_H)$  with a weight function  $w$  defined on its edges  $E_H$ . The weight of an edge  $[u, v]$  in  $E_H$  is defined to be equal to  $D_{\{G, w\}}(u, v)$ . The edges of  $H$  are defined as follows.

**Intra-cluster edges:** For all  $K_i$ , and for all  $u \in K_i$ ,  $[u, v_i] \in E_H$ .

**Inter-cluster edges:** For all  $v_i, v_j \in C$ ,  $[v_i, v_j]$  is an inter-cluster edge if either:

1.  $v_i \notin K_j$  and  $v_j \notin K_i$  and  $D_{\{G, w\}}(v_i, v_j) \leq R$  (Type 1), OR
2. there exists  $e = (u_i, u_j) \in E$  such that  $u_i \in K_i, u_j \in K_j$  (Type 2).

**Computing the Cluster Graph.** Here we describe how the cluster graph is computed efficiently under some assumptions. We first describe how a cluster-cover is computed. Once a cluster-cover is computed we show that the cluster graph can be easily computed. Note that the input to the cluster-cover computation is a weighted graph  $G(V, E)$  with a weight function  $w$  defined on its edges, a set  $C \subseteq V$  of cluster-centers and a radius  $R$ . We will assume that  $|V| = n$ ,  $|E| = O(n)$ , the weight function  $w$  is integral, and  $R$  is an integer. Since we do not have to deal with distances greater than  $R$ , we can safely assume that the weight of any edge is an integer value in the range  $[0..R]$ . We will further assume that the cluster-centers are chosen in such a way that a cluster-cover exists, which will be shown in Section 3.3.

The obvious way to implement this algorithm is as was done in [4], i.e., to run Dijkstra's SSSP algorithm from all the cluster-centers and to compute the clusters in the cluster-cover. However, this has a running time of  $O(n \log n)$ . In order to speed it up, we run Dijkstra's algorithm in **parallel** from all the cluster-centers and use a simple and faster priority queue, which we denote by  $PQ$ . The priority queue we use is an array of size  $R$ , indexed from 0 to  $R$ . This is sufficient for our purposes because of the following reasons. Firstly, the weight function is integral and the array contains all possible distance values from the cluster-centers to vertices in the clusters. Secondly, in Dijkstra's algorithm, once a vertex has been extracted from the priority queue, its distance from the source will never be updated again and the distance from the source at the time of the

extraction is the correct distance from the source. In other words, the minimum value of the items in the priority queue is monotonic. Since the priority queue is an array, EXTRACT-MIN can be implemented as a scan through the array for the “next” largest item. This means that the  $O(n)$  calls to EXTRACT-MIN needed by Dijkstra’s algorithm can be implemented in  $O(n + R)$  time.

One problem is that clusters can overlap and that vertices may have entries in the priority queue with distances from several cluster-centers. This can be taken care of by augmenting the priority queue entries to also store information about the vertex as well as the corresponding cluster-center. It should also be noted that Dijkstra’s algorithm needs to perform a number of RELAX steps and that in each such step the priority queue may need to be updated. The process of RELAXING an edge  $(u, v)$  consists of testing whether we can improve the shortest path to  $v$  found so far by going through  $u$  and, if so, updating the value for  $v$ . It should be pointed out that this is the only place where we are unable to eliminate the use of *Random Access* since it is critical that this update be performed efficiently, i.e., in  $O(1)$  time. Also note that an edge  $(u, v)$  may be RELAXED several times, each time with respect to a different cluster-center. Thus the time and space complexity of the algorithm is affected by the amount of overlap of the clusters in the cluster-cover. A careful implementation of cluster-cover can be made to run in time  $O(m \cdot c_v \cdot c_e + R)$ , where  $m$  is the number of cluster-centers,  $c_v$  is the maximum number of clusters that contain a vertex, and  $c_e$  is the maximum number of clusters that contain one of the endpoints of an edge. (In Section 3.3. we show that for our purposes  $c_v$  and  $c_e$  are constants.)

We now describe how to compute the cluster graph. The input is a weighted graph  $G$  with a weight function  $w$ , a set of cluster-centers  $C = \{v_1, \dots, v_m\}$ , and two different radii  $r$  and  $R$ . In order to compute the cluster graph, the algorithm computes a cluster-cover from the same set of cluster-centers but with the two radii,  $r$  and  $R$ . Let the cluster-covers with radii  $r$  and  $R$  be denoted by  $\mathcal{K}_r$  and  $\mathcal{K}_R$  respectively. We augment the cluster-cover procedure to also produce a data structure that supports the following queries for both the cluster-covers: (a) FINDCENTERS( $v, \mathcal{K}$ ): Given  $v \in V$ , it returns all cluster-centers  $v_i$  such that  $v$  is in a cluster from  $\mathcal{K}$  centered at  $v_i$ , i.e.,  $D_{G, Iwt}(v, v_i)$  is at most the radius of the clusters in  $\mathcal{K}$ . It also returns  $D_{G, w}(v, v_i)$  for these cluster-centers, and (b) COMPUTEDISTANCE( $v_i, v$ ): Given  $v \in V$ , and a cluster-center  $v_i$ , it returns  $D_{G, Iwt}(v, v_i)$  if  $D_{G, Iwt}(v, v_i) \leq R$ ; otherwise, it returns the value  $\infty$ .

Now the cluster graph  $H = (V, E_H)$  is computed easily as follows. The intra-cluster edges of  $H$  are computed by performing FINDCENTERS queries for each vertex  $v \in V$  in the cluster-cover  $\mathcal{K}_r$  and adding the corresponding edges. The inter-cluster edges can be of two types. An edge  $[v_i, v_j]$  of type 1 is added if  $v_i \notin K_j$  and  $v_j \notin K_i$  and  $D_{\{G, Iwt\}}(v_i, v_j) \leq R$ . Note that  $K_i$  and  $K_j$  are clusters of radii  $r$  with centers at  $v_i$  and  $v_j$  respectively. For every cluster-center  $v_i$ , we use the FINDCENTERS query to list all the clusters from  $\mathcal{K}_R$  that it is contained in. The centers  $v_j$  of these clusters satisfy the condition that  $D_{\{G, w\}}(v_i, v_j) \leq R$ . Now we use the COMPUTEDISTANCE queries to make sure that  $v_i \notin K_j$  and  $v_j \notin K_i$ . A careful consideration of all the steps above shows that the time

complexity of computing the cluster graph is  $O(n \cdot c_v^2)$ . Having the cluster centers before performing the clustering enables clusters to be grown in “parallel” and thus the above algorithm is able to use one common priority queue to grow all the clusters, and is consequently able to perform the clustering efficiently.

**Maintaining the Cluster Graph.** An edge of type 2 is added if there exists an edge  $e = (u_i, u_j) \in E$  such that  $u_i \in K_i$  and  $u_j \in K_j$ . During the computation of the cluster graph  $H$ , only intra-cluster edges and inter-cluster edges of type 1 are added. Additional edges may be added during a phase of the greedy algorithm. Every time the greedy algorithm decides to add an edge  $e = (u, v)$ , several inter-cluster edges of type 2 may be added to  $H$ . This is achieved as follows: for every edge  $e = (u_i, u_j)$  that is to be added to  $G'$ , perform FINDCENTERS queries for  $u_i$  and  $u_j$  from  $\mathcal{K}_r$  and join the corresponding cluster-centers by an inter-cluster edge in  $H$ . The weight of such edges are computed by performing two COMPUTEDISTANCE queries for  $u_i$  and  $u_j$  with the corresponding cluster-centers and adding it to  $w(u_i, u_j)$ . The above function runs in  $O(1)$  time.

**Selecting the Cluster Centers for a Phase.** In order for the CLUSTERGRAPH function to be implemented efficiently, it needs to have the set of cluster-centers as input. For the first phase, the cluster-centers  $C_1$  are identified in a greedy fashion using the weighted graph  $G' = (V, E_0)$  with real-valued edge weights, and using a radius of  $r$ . This is referred to as NAIVE-CENTERS in the algorithm given in Fig. 11. NAIVE-CENTERS runs in  $O(n \log n)$  time, since this can be implemented using the standard Dijkstra’s algorithm. For subsequent phases, cluster-centers are identified (using UPDATERCENTERS) in a different way. The set of cluster-centers are always chosen as a subset of the cluster-centers used in the previous phase. At the end of each phase, the algorithm selects a set of cluster centers for the next phase. These centers are guaranteed to be sufficiently far apart from each other. More specifically, the cluster centers  $C_i$  used in phase  $i$  are guaranteed to be at a distance of at least  $r/2$ .

In phase  $i$ , the set of cluster-centers for phase  $i + 1$  is computed as  $C_{i+1} := C_i \setminus M_i$ , i.e., a subset  $M_i$  of the cluster-centers are deleted from the list of cluster-centers. We now describe how the set  $M_i$  is chosen.  $M_1$  is the empty set, implying that  $C_2$  is identical to  $C_1$ . For  $i > 1$ , the algorithm picks a cluster-center from  $C_i$  and deletes all cluster-centers that are within distance  $r$  from it. (It is important to note that since the integralization changes in every iteration, vertices that are distance  $r'$  in one iteration are at distance  $r'/2$  in the next iteration.) This is easily implemented by using the FINDCENTERS query that is available after the cluster-cover for phase  $i$  has been computed. The next cluster-center is then picked and the process is continued until all centers are either picked or marked. Clearly this process runs in time  $O(m \cdot c_v)$ . We now show that in phase  $i$  the cluster-centers are guaranteed to be at a distance of at least  $r/2$  from each other. In phase 1, since cluster centers are identified by using a radius of  $r$ , all cluster centers are at a distance of at least  $r/2$  from each other. In phase  $i - 1$ , if two cluster-centers are at a distance of  $r$  or less, then one of them will get marked,

and will subsequently be deleted from the list  $C_i$  for phase  $i$ . Lemma 2 specifies conditions under which vertices belong to at most a constant number of clusters.

**Lemma 2.** *If  $C = \{v_1, v_2, \dots, v_m\} \subseteq V(G)$  vertices  $v_i, v_j \in C$ ,  $D_{\{G, Iwt\}}(v_i, v_j) > r'$ , and if  $\mathcal{K}_{r''} = \{K_1, K_2, \dots, K_m\}$ , is returned by CLUSTER-COVER( $G, C, r''$ ) and if  $r'' \leq c' \cdot r'$  for some constant  $c'$ , then each vertex  $v \in V(G)$  is contained in at most a constant (which depends on  $d$  and  $c'$ ) number of clusters from  $\mathcal{K}_{r'}$ .*

The conditions of the lemma are true for the cluster graph as constructed above with  $r' = r/2$  and  $c' = 2$  or  $c' = 4t/\delta$ . Hence any vertex in  $H$  is part of at most a constant number of clusters in  $\mathcal{K}_r$  or  $\mathcal{K}_R$ . The proof follows from standard packing arguments. Similar arguments also show that the number of inter-cluster edges incident on a cluster-center is also a constant (although it might have a large number of intra-cluster edges). It follows that the degree of any vertex in  $H$  that is not a cluster-center must be a constant, and the size of  $H$  is  $O(n)$ . Note that since the weights have been integralized, the resulting clusters are approximate clusters; they are a little bit larger (since integers are always rounded up) than the exact clusters.

### 3.3 Answering Shortest Path Queries

When the algorithm IMPROVED-GREEDY considers an edge  $e = (u, v)$  for inclusion in the spanner graph, it needs to answer a shortest path query. It needs to check if  $D_{\{G', wt\}}(u, v) \leq t \cdot d(u, v)$ , where  $G'$  is the spanner graph constructed so far. As noted earlier, it is sufficient for this query to be answered approximately. So, it is sufficient to devise a procedure to efficiently check if  $D_{\{G', wt\}}(u, v) \leq t(1 + \epsilon') \cdot d(u, v)$ , for some small  $\epsilon' > 0$ . In other words, it is sufficient to check if  $D_{\{G', Iwt\}}(u, v) \leq t(1 + \epsilon'') \cdot d(u, v)/U_i$ , for some small  $\epsilon'' > 0$ . In fact, the algorithm will check if  $D_{\{H, Iwt\}}(u, v) \leq t \cdot d(u, v)/U_i$ . The time complexity of this test is a constant if  $D_{\{H, Iwt\}}(u, v)$  is bounded by some constant multiple of  $r$  is a constant. Hence, we conclude this section by noting that the Improved-Greedy algorithm runs in time  $O(n \log n)$ .

### 3.4 The Graph Produced by IMPROVED-GREEDY Is a $t$ -Spanner.

In order to show that a valid cluster graph  $H$  approximates the graph  $G'$ , we need to prove lemmas that are analogous to Lemmas 3 and 4 from [4] modified to account for the error introduced by the integralization. Next, we need to show that the  $G'$  is a  $t$ -spanner for  $V$ . Since the clusters are computed using the function  $Iwt(\cdot)$  instead of  $wt(\cdot)$ , clusters are not as precise as they were in [4]. The following claims are stated without proof, which will be provided in a full version of the paper. Consider the cluster graph  $H$  that results from the clustering performed on  $G'$  at the start of phase  $i$ . The following claims apply to edges and paths in  $H$ . Many of them are modified versions of corresponding lemmas in [4]:

1. Let  $\mathcal{K}$  be equal to  $\text{CLUSTER}(G', v, r, Iwt)$  (i.e., it is a cluster with cluster-center  $v$  and radius  $r = \delta W$ ) computed in iteration  $i$  of the algorithm. If  $u$  is a vertex in  $\mathcal{K}$ , then  $D_{\{G', wt\}}(v, u) \leq (1 + \epsilon)rU_i$ . Otherwise, if  $u \notin \mathcal{K}$ , then  $D_{\{G', wt\}}(v, u) > rU_i$ .
2. If  $u$  is a cluster-center and  $[u, v]$  is an intra-cluster edge in the cluster-graph  $H$ , then  $D_{\{G', wt\}}(u, v) \leq (1 + \frac{\epsilon}{\delta})rU_i$ .
3. If  $[u, v]$  is an inter-cluster edge, then  $r_i < D_{\{G', wt\}}(u, v) \leq (1 + \epsilon) \cdot (R_i + 2r_i)U_i$ .
4. If there exists a path  $P_H$  in  $H$  between vertices  $u$  and  $v$  such that  $Iwt(P_H) = L$ , then there exists a path  $P_{G'}$  between  $u$  and  $v$  such that  $Iwt(P_{G'}) \leq L$ .
5. Let  $H$  be a valid Cluster graph of  $G'$  with cluster radii  $r$  and  $R = r/\delta$ . Let  $P_{G'}$  be a path between  $u$  and  $v$  in  $G'$  of weight  $Iwt(P_{G'})$  such that  $D_{\{G', wt\}}(u, v) > (1 + \epsilon)W - 2\delta W$ . Then there exists a path  $P_H$  between  $u$  and  $v$  in  $H$  such that  $Iwt(P_H) \leq \left(\frac{1+6\delta}{1-2\delta}\right) \cdot Iwt(P_{G'})$ .

The above claims are enough to prove that  $H$  is a cluster graph for  $G'$ , and consequently that  $G'$  is a  $t$ -spanner. As argued in Section 3.2, the resulting spanner graph has constant degree. The weight of the spanner is  $O(1) \cdot wt(MST)$  because of the *leapfrog property* from [4], the proof is omitted in this version.

This concludes the proof of Theorem 1 □

## 4 Conclusions and Acknowledgments

We present improved algorithms for the sparse spanner problem. In the process, we design linear-time algorithms for a clustering problem, which is likely to be of independent interest.

We are grateful to Professor Michiel Smid for helpful discussions and for pointing out errors in an earlier draft.

## References

1. S. Arya, G. Das, D. M. Mount, J. S. Salowe, and M. Smid. Euclidean spanners: short, thin, and lanky. In *Proc. ACM STOC'95*, pages 489–498, 1995.
2. P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to  $k$ -nearest-neighbors and  $n$ -body potential fields. *J. ACM*, 42:67–90, 1995.
3. D. Z. Chen, G. Das, and M. Smid. Lower bounds for computing geometric spanners and approximate shortest paths. In *Proc. CCCG'96*, pages 155–160, 1996.
4. G. Das and G. Narasimhan. A fast algorithm for constructing sparse Euclidean spanners. *Internat. J. Comput. Geom. Appl.*, 7(4):297–315, 1997.
5. J. M. Keil and C. A. Gutwin. Classes of graphs which approximate the complete Euclidean graph. *Discrete Comput. Geom.*, 7:13–28, 1992.
6. D. Mount. Personal communication, 1998.
7. F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.
8. S. B. Rao and W. D. Smith. Improved approximation schemes for traveling salesman tours. In *Proc. ACM STOC'98*, 1998.

9. J. S. Salowe. Construction of multidimensional spanner graphs with applications to minimum spanning trees. In *Proc. ACM SoCG'91*, pages 256–261, 1991.
10. M. Thorup. Undirected single-source shortest path with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.
11. P. M. Vaidya. A sparse graph almost as good as the complete graph on points in  $K$  dimensions. *Discrete Comput. Geom.*, 6:369–381, 1991.

# Computing the Penetration Depth of Two Convex Polytopes in 3D\*

Pankaj K. Agarwal<sup>1</sup>, Leonidas J. Guibas<sup>2</sup>, Sariel Har-Peled<sup>3</sup>,  
Alexander Rabinovitch<sup>4</sup>, and Micha Sharir<sup>5</sup>

<sup>1</sup> Center for Geometric Computing, Department of Computer Science, Box 90129,  
Duke University, Durham, NC 27708-0129, USA. tpankaj@cs.duke.edu

<sup>2</sup> Computer Graphics Laboratory, Computer Science Department, Stanford  
University, Stanford CA 94305 tguibas@cs.stanford.edu

<sup>3</sup> Center for Geometric Computing, Department of Computer Science, Box 90129,  
Duke University, Durham, NC 27708-0129, USA. tsariel@cs.duke.edu

<sup>4</sup> Synopsys Inc., 154 Crane Meadow Rd, Suite 300, Marlboro, MA 01752, USA.  
talexra@synopsys.com

<sup>5</sup> School of Mathematical Sciences, Tel Aviv University, Tel Aviv 69978, Israel; and  
Courant Institute of Mathematical Sciences, New York University, New York,  
NY 10012, USA. tsharir@math.tau.ac.il

**Abstract.** Let  $A$  and  $B$  be two convex polytopes in  $\mathbb{R}^3$  with  $m$  and  $n$  facets, respectively. The *penetration depth* of  $A$  and  $B$ , denoted as  $\pi(A, B)$ , is the minimum distance by which  $A$  has to be translated so that  $A$  and  $B$  do not intersect. We present a randomized algorithm that computes  $\pi(A, B)$  in  $O(m^{3/4+\varepsilon}n^{3/4+\varepsilon} + m^{1+\varepsilon} + n^{1+\varepsilon})$  expected time, for any constant  $\varepsilon > 0$ . It also computes a vector  $t$  such that  $\|t\| = \pi(A, B)$  and  $\text{int}(A + t) \cap B = \emptyset$ . We show that if the Minkowski sum  $B \oplus (-A)$  has  $K$  facets, then the expected running time of our algorithm is  $O(K^{1/2+\varepsilon}m^{1/4}n^{1/4} + m^{1+\varepsilon} + n^{1+\varepsilon})$ , for any  $\varepsilon > 0$ .

We also present an approximation algorithm for computing  $\pi(A, B)$ . For any  $\delta > 0$ , we can compute, in time  $O(m+n+(\log^2(m+n))/\delta)$ , a vector  $t$  such that  $\|t\| \leq (1+\delta)\pi(A, B)$  and  $\text{int}(A + t) \cap B = \emptyset$ . Our result also gives a  $\delta$ -approximation algorithm for computing the width of  $A$  in time  $O(n + (\log^2 n)/\delta)$ , which is simpler and slightly faster than the recent algorithm by Chan [4].

---

\* Work by P.A. was supported by Army Research Office MURI grant DAAH04-96-1-0013, by a Sloan fellowship, by NSF grants EIA-9870724, and CCR-9732787, and by a grant from the U.S.-Israeli Binational Science Foundation. Work by L.G. was supported in part by National Science Foundation grant CCR-9623851 and by US Army MURI grant 5-23542-A. Work by S.H.-P. was supported by the second author was supported by Army Research Office MURI grant DAAH04-96-1-0013. Work by M.S. was supported by NSF Grants CCR-97-32101, CCR-94-24398, by grants from the U.S.-Israeli Binational Science Foundation, the G.I.F., the German-Israeli Foundation for Scientific Research and Development, and the ESPRIT IV LTR project No. 21957 (CGAL), and by the Hermann Minkowski-MINERVA Center for Geometry at Tel Aviv University.

## 1 Introduction

Let  $A$  and  $B$  be two convex polytopes in  $\mathbb{R}^3$ , with  $m$  and  $n$  facets, respectively. The *penetration depth* of  $A$  and  $B$ , denoted as  $\pi(A, B)$ , is defined as

$$\pi(A, B) = \min\{\|t\| \mid \text{int}(A + t) \cap B = \emptyset, t \in \mathbb{R}^3\}.$$

One of the motivations for this problem comes from the field of robotics. Consider, for instance, the problem of collision detection in robot motion planning, where distance between objects is measured in the Euclidean metric. Numerous efficient algorithms are known for computing the minimum distance between two polyhedra in two and three dimensions (see [78]). Whenever two objects intersect, this distance measure is zero. Thus, it fails to provide any information about the *extent of penetration*. The penetration depth is a useful and natural measure of this extent. In addition, penetration depth can be a useful quantity to have available during physical simulations. Such simulations sample a moving system during discrete time steps and detect collisions between objects using a variety of methods. When a collision is detected, a penetration has usually occurred, because of the discrete time sampling. The penetration depth of the colliding bodies can be very useful in computing how to roll the simulation back to the instant of first contact, and in estimating the impulse force required for the appropriate collision response.

The problem is closely related to that of computing the *width* of a convex polytope  $A$ . Recall that the width of  $A$  is the shortest distance between any pair of parallel planes that support  $A$ . We will note below that if  $A = B$  then  $\pi(A, A) = \text{width}(A)$ . Thus the penetration depth is a natural extension of width. The best algorithm known for computing the width is by Agarwal and Sharir [2]; it is a randomized algorithm that runs in  $O(n^{3/2+\varepsilon})$  expected time, for any constant  $\varepsilon > 0$ . This algorithm is based on a randomized algorithm, presented in [2], for computing the closest bichromatic pair of lines for two vertically-separated sets  $L$  and  $L'$  of lines in  $\mathbb{R}^3$  in expected time  $O(|L|^{3/4+\varepsilon}|L'|^{3/4+\varepsilon} + |L|^{1+\varepsilon} + |L'|^{1+\varepsilon})$ , for any  $\varepsilon > 0$ . We use this algorithm for computing  $\pi(A, B)$  in expected time  $O(m^{3/4+\varepsilon}n^{3/4+\varepsilon} + m^{1+\varepsilon} + n^{1+\varepsilon})$ , for any  $\varepsilon > 0$ . Actually, we will show that if the number of facets of the Minkowski sum  $B \oplus (-A)$  is  $K$ , then the expected running time of the algorithm is  $O(K^{1/2+\varepsilon}m^{1/4}n^{1/4} + m^{1+\varepsilon} + n^{1+\varepsilon})$ , for any  $\varepsilon > 0$ . This is, to the best of our knowledge, the first subquadratic algorithm for computing  $\pi(A, B)$ .

Dobkin et al. [10] showed that  $A$  and  $B$  can be preprocessed in  $O(m+n)$  time so that, for a direction  $u$ , the distance by which  $A$  has to be translated in direction  $u$  to separate it from  $B$ , denoted as  $\Delta(u)$ , can be computed in  $O(\log^2(m+n))$  time. We use this result to obtain a simple approximation algorithm for computing  $\pi(A, B)$ . In particular, for any given  $\delta > 0$ , we present an  $O(m+n+(\log^2(m+n))/\delta)$ -time algorithm for computing a vector  $u$  such that  $\text{int}(A+u) \cap B = \emptyset$  and  $\|u\| \leq (1+\delta)\pi(A, B)$ .

Our results imply an “output-sensitive” algorithm for computing the width of a convex polytope  $A$  with  $n$  facets in randomized expected time  $O(K^{1/2+\varepsilon}n^{1/2} +$

$n^{1+\varepsilon}$ ), where  $K$  is the number of facets in  $A \oplus (-A)$ , and a  $(1+\delta)$ -approximation algorithm for computing the width of  $A$  in time  $O(n + (\log^2 n)/\delta)$ . This approximation algorithm is simpler and slightly faster than the recent algorithm by Chan [4], which computes a  $(1 + \delta)$ -approximation of  $\text{width}(A)$  in time  $O(n + (\log^c n)/\delta)$  for some constant  $c > 2$ . Finally, we show how the Dobkin-Kirkpatrick hierarchical representations of two convex polytopes  $A$  and  $B$  can be used to obtain efficient implementation of various extremal queries concerning the Minkowski sum  $B \oplus (-A)$  without its explicit construction. For lack of space, we omit this part from the current abstract.

## 2 Computing the Penetration Depth

Before describing the algorithm for computing  $\pi(A, B)$ , we note the relationship between the penetration depth of two polytopes and the width of a polytope.

**Proposition 1** *For any convex polytope  $P$  in  $\mathbb{R}^3$ ,  $\text{width}(P) = \pi(P, P)$ .*

*Proof.* Let  $\lambda$  denote the length of the shortest translation vector that separates two initially-identical copies of  $P$ . Let  $v$  be the vector realizing the width of  $P$ ; that is,  $v$  is a shortest vector between two parallel supporting planes of  $P$  that realize the width of  $P$ . Clearly,  $\text{int}(P + v) \cap \text{int}(P) = \emptyset$ , and therefore  $\lambda \leq \|v\| = \text{width}(P)$ . As for the other direction, let  $u$  be a shortest separating translation vector. Clearly,  $P$  and  $P + u$  touch each other but have disjoint interiors. Thus, there is a plane  $H$  that separates the interiors of  $P$  and  $P + u$ , and intersects both  $P$  and  $P + u$ . In particular,  $P$  lies between the two planes  $H$  and  $H - u$ . Since the distance between  $H$  and  $H - u$  is at most  $\|u\|$ , it follows that

$$\text{width}(P) \leq d(H, H - u) \leq \|u\| = \lambda.$$

This proposition suggests that we attempt to modify the width-algorithm by Agarwal and Sharir [2] to compute  $\pi(A, B)$ , which is indeed what we proceed to do. Conversely, we will also specialize the new techniques developed in this paper to obtain new approximation and output-sensitive algorithms for computing  $\text{width}(A)$ .

*Overall algorithm.* Let  $A$  and  $B$  be two convex polytopes as defined above. Using linear programming, we can determine in  $O(m + n)$  time whether  $A$  and  $B$  intersect [3]. If  $A$  and  $B$  do not intersect, then we set  $\pi(A, B) = 0$  and stop. So we assume that  $A \cap B \neq \emptyset$ .

We can formulate the problem of computing  $\pi(A, B)$  in terms of the *configuration space* that represents all possible placements of  $A$  relative to (the fixed)  $B$ . That is,  $A$  turns into a point  $p(A)$  and  $B$  turns into the *Minkowski sum*  $B \oplus (-A) = \{x - y \mid x \in B, y \in A\}$ . Let us assume that the initial location of the point  $p(A)$  corresponding to  $A$  in the configuration space is the origin  $O$  of the coordinate system. Note that  $p(A)$  is inside the polytope  $\mathcal{P} = B \oplus (-A)$  if

and only if  $A$  (in the corresponding translated placement) and  $B$  intersect. By construction, it follows that

$$\pi(A, B) = \min\{d(O, x) \mid x \in \partial P\}.$$

Let  $x \in \partial P$  be a point on the boundary so that  $d(x, O) = d(O, P)$ . Then  $\overrightarrow{Ox}$  is orthogonal to the facet of  $P$  containing  $x$ , because otherwise we could obtain an even shorter distance from  $O$  to  $\partial P$ , which is impossible (alternatively, the penetration distance is the smallest radius of a ball  $B_r$ , centered at the origin, that touches the boundary of  $P$ ). Therefore,  $d(x, O)$  is attained as a shortest distance between  $O$  and a plane that contains the corresponding facet of  $P$ . In particular, we can compute the penetration distance by computing the distance between the origin and all those planes.

Every facet of  $P$  is attained as a Minkowski sum of the form  $g \oplus (-f)$ , where  $g$  is a facet, edge, or vertex of  $B$  and  $f$  is, respectively, a vertex, edge, or facet of  $A$ . It is well known that there are only  $O(m + n)$  facets of  $P$  for which  $g$  is a facet or a vertex of  $B$  (and  $f$  is a vertex or a facet of  $A$ ), and they can all be found in  $O((m + n) \log(m + n))$  time (see e.g. [5]). Hence, determining the minimum distance from  $O$  to all these facets can be done in near-linear time. The problem is to handle facets that are attained as Minkowski sums of pairs of edges of the form  $e \oplus (-e')$  such that  $e$  is an edge of  $B$  and  $e'$  is an edge of  $A$ . In the worst case, there can be  $\Omega(mn)$  such facets. However, not every such pair necessarily generate a facet of  $P$ .

We partition the edges of  $A$  and  $B$  into a family of pairs of subsets of edges

$$\mathcal{F} = \{(A_1, B_1), \dots, (A_u, B_u)\}$$

such that the following five conditions hold.

- (C1)  $A_i$  (resp.  $B_i$ ) is a subset of the edges of  $A$  (resp.  $B$ ).
- (C2) Every pair  $(e', e) \in A_i \times B_i$  generates a facet of  $P$ .
- (C3) Every pair of edges that generate a facet of  $P$  appears in some  $A_j \times B_j$ .
- (C4) For each  $i$ , the lines supporting the edges in  $A_i$  and those in  $B_i$  are vertically separated. That is, either all lines supporting the edges of  $A_i$  lie above all lines supporting the edges of  $B_i$ , or all of them lie below the lines supporting the edges of  $B_i$ .
- (C5)  $\mathcal{F}$  can be partitioned into two subfamilies  $\mathcal{F}^A$  and  $\mathcal{F}^B$  such that
  - (i) for every  $0 \leq i \leq \lfloor \log_2 m \rfloor$ , there are  $O((m/2^i) \log m)$  pairs  $(A_i, B_i)$  in  $\mathcal{F}^A$  for which  $2^i \leq |A_i| < 2^{i+1}$ . Let  $\mathcal{F}_i^A$  denote the subset of these pairs. Then  $\sum_{(A_j, B_j) \in \mathcal{F}_i^A} |B_j| = O(n \log n)$ ; and
  - (ii) for every  $0 \leq i \leq \lfloor \log_2 n \rfloor$ , there are  $O((n/2^i) \log n)$  pairs  $(A_i, B_i)$  in  $\mathcal{F}^B$  for which  $2^i \leq |B_i| < 2^{i+1}$ . Let  $\mathcal{F}_i^B$  denote the subset of these pairs. Then  $\sum_{(A_j, B_j) \in \mathcal{F}_i^B} |A_j| = O(m \log m)$ .

Suppose we have such a decomposition at our disposal. Then we can compute  $\pi(A, B)$  as follows.

Algorithm: PENETRATION-DEPTH ( $A, B$ )

1. For each pair  $(f, g)$  such that  $f$  is a vertex or facet of  $A$  and  $g$  is a facet of  $B$ , and  $g \oplus (-f)$  is a facet of  $\mathcal{P}$ , compute the distance from the origin to the plane containing  $g \oplus (-f)$ . Let  $\Delta^*$  be the minimum of these distances.
2. For each pair  $(A_i, B_i)$  in the above decomposition, find the minimum distance  $\Delta_i$  from the origin to an element in the set of planes

$$H_i = \{\text{aff}(e \oplus (-e')) \mid e \in B_i, e' \in A_i\},$$

where  $\text{aff}(e \oplus (-e'))$  is the plane containing the facet of  $B \oplus (-A)$  induced by  $e$  and  $e'$ .

3. Return  $\min\{\Delta^*, \min_i\{\Delta_i\}\}$ .

The correctness of this algorithm is obvious. Step 1 considers all facets of  $\mathcal{P}$  induced by a vertex-facet pair of  $A$  and  $B$ . By Condition (C2), the algorithm considers only those pairs of edges that generate facets of  $\mathcal{P}$ , and by Condition (C3), the algorithm considers all such pairs. It thus suffices to show how to compute  $\Delta_i$ , for each pair  $(A_i, B_i)$ , and how to construct the family  $\mathcal{F}$ .

*Computing  $\Delta_i$ .* Let  $(A_i, B_i)$  be a pair in  $\mathcal{F}$ . Denote by  $L_i$  and  $L'_i$ , respectively, the sets of lines that contain the edges of  $B_i$  and  $A_i$ .

**Lemma 2.** *For any pair  $(A_i, B_i) \in \mathcal{F}$ ,  $\Delta_i = d(L_i, L'_i)$ .*

*Proof.* Let  $e \in B_i$  and  $e' \in A_i$ , and let  $\ell$  and  $\ell'$  be the lines that contain  $e$  and  $e'$ , respectively. Consider the the plane  $h = \ell \oplus (-\ell') = \{x - y \mid x \in \ell, y \in \ell'\}$ . Note that for any two sets  $X$  and  $Y$ ,

$$d(O, X \oplus (-Y)) = \inf\{|x - y| \mid x \in X, y \in Y\} = d(X, Y).$$

Therefore  $d(O, h) = d(O, \ell \oplus (-\ell')) = d(\ell, \ell')$ . Thus,

$$\begin{aligned} \Delta_i &\equiv \min_{h \in H_i} d(O, h) \\ &= \min\{d(O, \ell \oplus (-\ell')) \mid \ell \in L_i, \ell' \in L'_i\} \\ &= \min\{d(\ell, \ell') \mid \ell \in L_i, \ell' \in L'_i\} \\ &= d(L_i, L'_i). \end{aligned}$$

By the above lemma, computing  $\Delta_i$  reduces to computing a closest bichromatic pair of lines in  $L_i \times L'_i$ . Recall that by Condition (C4) on  $\mathcal{F}$ , the lines in  $L_i$  and  $L'_i$  are vertically separated. Agarwal and Sharir [2] showed that under this condition, the closest pair in  $L_i \times L'_i$  can be computed in expected time  $O(|L_i|^{3/4+\varepsilon}|L'_i|^{3/4+\varepsilon} + |L_i|^{1+\varepsilon} + |L'_i|^{1+\varepsilon})$ . Summing this bound over all pairs in  $\mathcal{F}$  and using property (C5), we can prove that the total time spent in computing all  $\Delta_i$ 's is  $O(m^{3/4+\varepsilon}n^{3/4+\varepsilon} + m^{1+\varepsilon} + n^{1+\varepsilon})$ , for any  $\varepsilon > 0$ .

*Computing  $\mathcal{F}$ .* Our decomposition is based on the following observation. Let  $\mathcal{M}$  denote the *Gaussian diagram* (or *normal diagram*) of  $B$ .  $\mathcal{M}$  is a spherical map on the unit sphere  $S^2$ . The vertices of  $\mathcal{M}$  are points on  $S^2$ , each representing the direction of the outward normal of a facet of  $B$ , the edges of  $\mathcal{M}$  are great circular arcs, each being the locus of the outward normal directions of all planes supporting  $B$  at some fixed edge, and the faces of  $\mathcal{M}$  are regions, each being the locus of outward normal directions of all planes supporting  $B$  at a vertex.  $\mathcal{M}$  can be computed in linear time from  $B$ . Let  $\mathcal{M}'$  be the similarly-defined normal diagram of  $-A$ . Consider the superposition of  $\mathcal{M}$  and  $\mathcal{M}'$ . Each intersection point between an arc of  $\mathcal{M}$  and an arc of  $\mathcal{M}'$ , representing respectively an edge  $e$  of  $B$  and an edge  $e'$  of  $A$ , gives us a direction  $\mathbf{u}$  which is orthogonal to the plane containing the Minkowski sum  $e \oplus (-e')$ . Furthermore,  $e \oplus (-e')$  is a real facet of  $B \oplus (-A)$ . It follows that a pair of edges of  $A$  and  $B$  generates a face of  $B \oplus (-A)$  if and only if the corresponding arcs intersect in the overlapped diagram. Note that the number of such arc intersections on this diagram can be  $\Omega(nm)$ .

Our goal is thus to decompose all pairs of intersecting arcs of  $\mathcal{M}$  and  $\mathcal{M}'$ . Without loss of generality, assume that no intersection point of  $\mathcal{M}$  and  $\mathcal{M}'$  lies on the equator. (We can either handle these intersections separately, or perform a random simultaneous rotation on  $\mathcal{M}$  and  $\mathcal{M}'$ .) If an arc of  $\mathcal{M}$  or  $\mathcal{M}'$  crosses the equator, we split it into two by adding a vertex on the arc at the equator. Hence each arc lies completely in the upper or the lower hemisphere. Let  $\mathbb{H}$  denote the upper hemisphere of  $S^2$ . We will describe how we decompose the set of edges of  $A$  and  $B$  whose corresponding arcs intersect in  $\mathbb{H}$ ; the lower hemisphere is handled similarly.

Note that the arcs in  $\mathcal{M}$  (and in  $\mathcal{M}'$ ) are pairwise disjoint. We centrally project the arcs of  $\mathcal{M}$  and  $\mathcal{M}'$  that lie in  $\mathbb{H}$  onto the plane  $h : z = 1$ . Since each arc of  $\mathcal{M}$  and  $\mathcal{M}'$  is a portion of a great circle, it projects to a segment (or a ray) on  $h$ . Let  $E$  (resp.  $E'$ ) be the set of projected segments of arcs in  $\mathcal{M}$  (resp.  $\mathcal{M}'$ ). By construction, the interiors of the segments in  $E$  (or  $E'$ ) are pairwise disjoint.

As described in [6], we decompose the set of intersecting pairs of segments in  $E$  and  $E'$  into a family  $\mathcal{F}' = \{(E_1, E'_1), \dots, (E_u, E'_u)\}$  as follows. We construct two segment trees  $T_A$  and  $T_B$  on the segments of  $E$  and  $E'$ , respectively. Each node  $v$  of  $T_A$  (resp.  $T_B$ ) corresponds to a vertical strip, with an associated subset  $E_v \subseteq E$  (resp.  $E'_v \subseteq E'$ ) that completely cross the strip. For each such subset, we construct a balanced binary tree, sorted by the height of those segments inside the strip (the segments are nonintersecting, and thus the ordering is well defined). For each node  $w$  of this binary tree, we refer to the subset of segments stored in the subtree rooted at  $w$  as a *canonical subset*.

For each segment  $e$  of  $E'$  (resp.  $E$ ), we find the nodes  $v$  of  $T_A$  (resp.  $T_B$ ) such that at least one endpoint of  $e$  lies inside the strip associated with the parent of  $v$ ; there is a logarithmic number of such nodes. We report all segments of  $E_v$  (resp.  $E'_v$ ) intersected by the segment as the union of a logarithmic number of canonical subsets. After repeating this step for all segments, for each canonical

subset  $E_w$  of  $T_A$ , we report the pair  $(E_w, E'_w)$ , where  $E'_w$  is the subset of segments for which the query procedure returned  $E_w$  as one of the canonical subsets. We do the same for the canonical subsets of  $T_B$ . It is shown in [5] that if segments  $e \in E, e' \in E'$  intersect, then there is one such pair  $(E_z, E'_z)$  such that  $e \in E_z$  and  $e' \in E'_z$ , and that the total time spent is  $O((m+n) \log(m+n))$ . Finally, for each pair  $(E_w, E'_w)$ , let  $A_w$  (resp.  $B_w$ ) be the set of corresponding edges of  $A$  and  $B$ . We add the pair  $(A_w, B_w)$  to  $\mathcal{F}$ .  $\mathcal{F}^A$  (resp.  $\mathcal{F}^B$ ) is the subset of pairs corresponding to the canonical subsets of  $T_A$  (resp.  $T_B$ ). The argument in [5] shows that  $\mathcal{F}$  satisfies conditions (C1)–(C3) and (C5). Condition (C4) follows from the following lemma.

**Lemma 3.** *Let  $e$  be an edge of  $B$  and  $e'$  an edge of  $A$  such that the corresponding arcs intersect in  $\mathbb{H}$ . Then the line supporting  $e$  lies above the line supporting  $e'$ .*

*Proof.* Since the arcs corresponding to  $e$  and  $e'$  intersect in  $\mathbb{H}$ , the sum  $e \oplus (-e')$  is a facet of  $B \oplus (-A)$  with an outward normal direction  $u$  that points upwards. By construction of the diagrams, there are planes  $h, h'$  orthogonal to  $u$  and supporting, respectively,  $B$  at  $e$  and  $A$  at  $e'$ . Moreover, relative to the direction  $u$ ,  $B$  lies below  $h$  and  $A$  lies above  $h'$ . It follows that since  $A$  and  $B$  intersect, the plane  $h$  is above the plane  $h'$  relative to the direction  $u$ . Thus also the line  $\ell$  containing  $e$  is above the line  $\ell'$  containing  $e'$  relative to the direction  $u$ . We assume that the vertices of  $A$  and  $B$  are in general position. In particular, there is no four vertices lying in the same plane. Thus the lines  $\ell$  and  $\ell'$  are not parallel. Let  $\ell_0$  be the unique upward-directed vertical line that passes through  $\ell$  and  $\ell'$ . Since the angle between  $\ell_0$  and  $u$  is smaller than  $\pi/2$ , and a line in direction  $u$  crosses  $h'$  before  $h$ , it follows that  $\ell_0$  also crosses  $h'$  (at a point on  $\ell'$ ) before it crosses  $h$  (at a point on  $\ell$ ). Hence  $\ell$  lies vertically above  $\ell'$ , as claimed.

Hence, we conclude the following.

**Theorem 1.** *Given two convex polytopes  $A, B$  in  $\mathbb{R}^3$  with  $m$  and  $n$  vertices, respectively, the penetration depth of  $A$  and  $B$  can be computed in (randomized expected) time  $O(m^{3/4+\varepsilon}n^{3/4+\varepsilon} + m^{1+\varepsilon} + n^{1+\varepsilon})$ , for any  $\varepsilon > 0$ ; the constant of proportionality depends on  $\varepsilon$ .*

*An output-sensitive bound.* Let  $K$  denote the number of facets in  $\mathcal{P} = B \oplus (-A)$ . We derive a bound on the expected running time of the algorithm that depends on  $K$ . Note that the pair  $(A_i, B_i)$  contributes  $|A_i| \cdot |B_i|$  facets to  $\mathcal{P}$ . The expected running time of the algorithm is

$$\begin{aligned} & \sum_{i=1}^u O\left((|A_i||B_i|)^{3/4+\varepsilon} + |A_i|^{1+\varepsilon} + |B_i|^{1+\varepsilon}\right) \\ &= O\left(K^\varepsilon \sum_{i=1}^u (|A_i||B_i|)^{3/4} + m^{1+\varepsilon} + n^{1+\varepsilon}\right). \end{aligned}$$

We obtain a bound on  $\sum_{(A_j, B_j) \in \mathcal{F}^A} (|A_j||B_j|)^{3/4}$ . A similar argument bounds the quantity for pairs in  $\mathcal{F}^B$ . Let  $K_i$  be the number of facets contributed by the

pairs in  $\mathcal{F}_i^A$ . Recall that  $|\mathcal{F}_i^A| = O(m/2^i \log m)$ .

$$\begin{aligned} \sum_{(A_j, B_j) \in \mathcal{F}^A} (|A_j| \cdot |B_j|)^{3/4} &= \sum_{i=0}^{\log_2 m} \sum_{(A_j, B_j) \in \mathcal{F}_i^A} (|A_j| \cdot |B_j|)^{3/4} \\ &\leq \sum_{i=0}^{\log_2 m} 2^{3(i+1)/4} \left( \sum_{(A_j, B_j) \in \mathcal{F}_i^A} |B_j| \right)^{3/4} |\mathcal{F}_i^A|^{1/4} \\ &\leq \sum_{i=0}^{\log_2 m} 2^{3i/4} O\left(\left(\frac{K_i}{2^i}\right)^{3/4}\right) \cdot \left(\frac{m \log m}{2^i}\right)^{1/4} \\ &\leq (m \log m)^{1/4} \sum_{i=0}^{\log_2 m} O\left(\frac{K_i^{3/4}}{2^{i/4}}\right), \end{aligned}$$

where the second last inequality follow from the fact that

$$K_i = \sum_{(A_j, B_j) \in \mathcal{F}_i^A} |A_j| |B_j| \geq 2^i \sum_{(A_j, B_j) \in \mathcal{F}_i^A} |B_j|.$$

On the other hand,  $K_i \leq 2^{i+1} \sum_{(A_j, B_j) \in \mathcal{F}_i^A} |B_j| \leq c2^i n \log n$  for a constant  $c > 1$ . The term  $\sum_i K_i^{3/4}/2^{i/4}$  is therefore maximized when  $K_i = c2^i n \log n$  for  $0 \leq i \leq \log_2 \frac{K}{cn \log n}$  and 0 otherwise. Hence,

$$\sum_{i=0}^{\log_2 m} O\left(\frac{K_i^{3/4}}{2^{i/4}}\right) = \sum_{i=0}^{\log_2 \frac{K}{cn \log n}} O(2^{i/2} (n \log n)^{3/4}) = O(\sqrt{K} (n \log n)^{1/4}).$$

Therefore  $\sum_{(A_j, B_j) \in \mathcal{F}^A} (|A_j| \cdot |B_j|)^{3/4} = O(\sqrt{K} (mn \log n)^{1/4})$ , which implies the following.

**Theorem 2.** *Given two intersecting convex polytopes  $A, B$  in  $\mathbb{R}^3$  with  $m$  and  $n$  vertices, respectively, such that  $B \oplus (-A)$  has  $K$  facets, one can compute the penetration depth of  $A$  and  $B$  in (randomized expected) time  $O(K^{1/2+\varepsilon} m^{1/4} n^{1/4} + m^{1+\varepsilon} + n^{1+\varepsilon})$  for any  $\varepsilon > 0$ .*

An immediate corollary of the above theorem is the following.

**Corollary 4** *Given a convex polytope  $A$  in  $\mathbb{R}^3$  with  $n$  vertices such that  $A \oplus (-A)$  has  $K$  facets, one can compute the width of  $A$  in (randomized expected) time  $O(K^{1/2+\varepsilon} \sqrt{n} + n^{1+\varepsilon})$  for any  $\varepsilon > 0$ .*

### 3 An Approximation Algorithm

We now present an efficient algorithm for approximating the penetration depth of  $A$  and  $B$ . That is, for a given  $\delta > 0$ , the algorithm computes a translation

vector  $t$  such that  $A + t$  and  $B$  are disjoint and  $\|t\| \leq (1 + \delta)\pi(A, B)$ . The algorithm is as follows.

Algorithm: APPROX-SEPARATION ( $A, B$ )

1. Define on the unit sphere of directions a grid  $G$  of points in the following manner: Divide the interval of angles  $[0, \pi]$  into  $\lceil c_1/\sqrt{\delta} \rceil$  subintervals of equal length, delimited by the points  $0 = p_0, p_1, \dots, p_{\lceil c_1/\sqrt{\delta} \rceil} = \pi$ , where  $c_1$  is a constant independent of  $\delta$  which will be specified later. Then the grid  $G$  is defined as the set of points

$$G = \{(p_i, 2p_j) \mid 0 \leq i, j \leq \lceil c_1/\sqrt{\delta} \rceil\},$$

where the points are given in spherical coordinates  $(\varphi, \theta)$ .

2. For each point  $p \in G$  apply the following procedure: Perform in the configuration space a ray-shooting query from the origin in the direction  $Op$ . Let  $\Delta(p)$  be the Euclidean distance from  $O$  to the boundary of  $B \oplus (-A)$  in this direction. (We will explain below how the ray-shooting can be performed efficiently without explicit computation of the Minkowski sum.)
3. Output  $\Delta = \min_{p \in G} \{\Delta(p)\}$  as an approximate solution.

**Lemma 5.** *For any  $\delta > 0$ , algorithm APPROX-SEPARATION computes correctly a translation of length  $\Delta$  that separates  $A$  and  $B$ , such that  $\Delta \leq (1 + \delta)\pi(A, B)$ .*

*Proof.* See [1, Section 3].

The size of the grid built by the APPROX-SEPARATION algorithm is  $O(1/\delta)$ . It was shown by Dobkin et al. [10], that after a linear-time preprocessing of  $A$  and  $B$  into suitable data structures, the shortest separation of  $A$  and  $B$  along any query direction  $u$  can be computed in time  $O(\log^2(m+n))$ . This operation is equivalent to performing a ray shooting in the direction  $u$  from the origin towards  $\partial(\mathcal{P})$ . Therefore, the total running time of the algorithm APPROX-SEPARATION is  $O(m + n + (\log^2(m+n))/\delta)$ . We have thus shown:

**Theorem 3.** *Given two convex polytopes  $A$  and  $B$  in  $\mathbb{R}^3$ , with  $m$  and  $n$  facets, respectively, and a parameter  $\delta > 0$ , one can compute, in time  $O(m + n + (\log^2(m+n))/\delta)$ , a separating translation for  $A$  and  $B$  whose length is at most  $(1 + \delta)\pi(A, B)$ .*

Applying Proposition 1, we also obtain the following corollary:

**Corollary 6** *For any  $\delta > 0$ , a  $(1 + \delta)$ -approximation of the width of a convex polytope in  $\mathbb{R}^3$  with  $n$  facets can be computed in time  $O(n + (\log^2 n)/\delta)$ .*

*Penetration depth under polyhedral metric.* Another application of this technique is to obtain a linear-time algorithm for computing the penetration depth of of  $A$  and  $B$  under any polyhedral norm (whose unit ball is a polytope with  $O(1)$  facets, such as the  $L^1$  or  $L^\infty$  norms). Let  $Q$  be a centrally-symmetric convex polytope with  $O(1)$  facets, and let  $\|\cdot\|_Q$  denote the norm induced by  $Q$ . We observe that the  $\|\cdot\|_Q$ -distance from  $O$  to the boundary of  $\mathcal{P}$  is equal to the largest scaling factor  $\lambda$  such that  $\lambda Q \subseteq \mathcal{P}$ . As is easily seen, a vertex of  $\lambda Q$  must then touch  $\partial(\mathcal{P})$ . Moreover, when  $\lambda$  varies, each vertex of  $\lambda Q$  traces a ray from the origin. Hence, to find the largest  $\lambda$ , we perform ray shootings from  $O$  in each of the  $O(1)$  directions of the vertices of  $Q$ . For each of these ray shootings we compute the scaling  $\lambda$  that corresponds to the hitting point of that ray with  $\partial\mathcal{P}$ . The smallest of these  $\lambda$ 's is the desired  $\|\cdot\|_Q$ -length of the shortest separating translation. We have thus shown:

**Corollary 7** *The shortest separating translation of two convex polytopes  $A$  and  $B$  in  $\mathbb{R}^3$ , with  $m$  and  $n$  facets, respectively, under any polyhedral norm (whose unit ball has  $O(1)$  facets) can be computed in  $O(m + n)$  time.*

*Handling shallow penetrations.* If the penetration of  $A$  into  $B$  is relatively small, then one might expect that the following combinatorial property holds in practice. Let  $\delta > 0$  be a small parameter. Then the number of facets of  $\mathcal{P} = B \oplus (-A)$  whose distance from the origin is at most  $(1 + \delta)\pi(A, B)$  is small, say  $K_\delta$ . If this is the case, then the following more efficient algorithm computes  $\pi(A, B)$ .

Algorithm: SHALLOW-PENETRATION ( $A, B$ )

1. Construct the grid  $G$  as in Algorithm APPROX-SEPARATION.
2. Using Algorithm APPROX-SEPARATION, compute a real value  $\Delta$  such that  $\Delta \leq (1 + \delta/4)\pi(A, B)$ .
3. Compute  $G' = \{u \in G \mid \Delta(u) \leq (1 + \delta/4)\Delta\}$ .
4. Let  $\mathcal{B}$  be the ball of radius  $(1 + \delta/2)\Delta \leq (1 + \delta)\pi(A, B)$  centered at the origin.
5. For each  $u \in G'$ , do the following:
  - (i) Compute the face  $f$  of  $\mathcal{P}$  supported by the plane orthogonal to  $u$ .
  - (ii) By performing an implicit breadth-first search on  $\partial\mathcal{P}$ , compute the connected component  $C_u$  of  $(\partial\mathcal{P}) \cap \mathcal{B}$  that contains  $f$ . (If  $C_u = C_v$  for two directions  $u \neq v$ , we compute the connected component  $C_u$  only once.)
  - (iii) Compute  $\Delta_u = \min_{f \in C_u} d(O, f)$ , where  $f$  is a facet of  $\mathcal{P}$  in  $C_u$ .
6. Return  $\min_{u \in G'} \Delta_u$ .

Steps (1)–(3) can be performed in  $O(m + n + (\log^2(m + n))/\delta)$  time as described in the algorithm APPROX-SEPARATION. For a given  $u \in G'$ , we can compute  $C_u$  in  $O((1 + |C_u|)\log(m + n))$  time by locating  $u$  in  $\mathcal{M}$  and  $\mathcal{M}'$  and by traversing the two normal diagrams simultaneously. We omit the easy details from this abstract. Computing  $\Delta_u$  takes  $O(|C_u|)$  time. Since we traverse each connected component of  $(\partial\mathcal{P}) \cap \mathcal{B}$  at most once, the total time spent in Step (5) is  $O((K_\delta + 1/\delta)\log(m + n))$ , where  $K_\delta$  is the number of facets of  $\mathcal{P}$  that lies

within distance  $(1 + \delta)\pi(A, B)$  from  $O$ . The same argument as in Lemma 5 can be used to show that the above algorithm computes all those connected components of  $(\partial\mathcal{P}) \cap \mathcal{B}$  that contain a facet within distance  $\pi(A, B)$  from  $O$ . Hence,  $\min_{u \in G'} \Delta_u = \pi(A, B)$ . We thus obtain the following.

**Theorem 4.** *Given two convex polytopes  $A$  and  $B$  in  $\mathbb{R}^3$ , with  $m$  and  $n$  facets, respectively, and a parameter  $\delta > 0$ , one can compute  $\pi(A, B)$  in time  $O(m+n+K_\delta \log(m+n) + (\log^2(m+n))/\delta)$ , where  $K_\delta$  is the number of facets of  $B \oplus (-A)$  within distance  $(1 + \delta)\pi(A, B)$  from the origin.*

## References

1. P.K. Agarwal, S. Har-Peled, M. Sharir, and K. R. Varadarajan. Approximate shortest paths on a convex polytope in three dimensions. *J. Assoc. Comput. Mach.*, 44:567–584, 1997.
2. P. K. Agarwal and M. Sharir, Efficient randomized algorithms for some geometric optimization problems, *Discrete Comp. Geom.* 16 (1996), 317–337.
3. M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, Berlin-Heidelberg, 1997, pp. 29–33.
4. T. Chan, Approximating the diameter, width, smallest enclosing cylinder, and minimum-width annulus, to appear in *Proc. 16th ACM Sympos. Comput. Geom.*, 2000.
5. B. Chazelle, H. Edelsbrunner, L. Guibas and M. Sharir, Algorithms for bichromatic line segment problems and polyhedral terrains, *Algorithmica*, 11 (1994), 116–132.
6. B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir, Diameter, width, closest line pair and parametric searching, *Discrete Comput. Geom.* 10 (1993), 183–196.
7. F. Chin and C.A. Wang, Optimal algorithms for the intersection and the minimum distance problems between planar polygons, *IEEE Trans. on Computers*, 32 (1983), 1203–1207.
8. D. Dobkin and D. Kirkpatrick, A linear algorithm for determining the separation of convex polyhedra, *J. of Algorithms*, 6 (1985), 381–392.
9. D. Dobkin and D. Kirkpatrick, Determining the separation of preprocessed polyhedra—a unified approach, *Proc. ICALP '90*, 400–413. Lecture Notes in Computer Science, Vol. 443, Springer-Verlag, Berlin, 1990.
10. D. Dobkin, J. Hershberger, D. Kirkpatrick, and S. Suri, Computing the intersection-depth of polyhedra, *Algorithmica* 9 (1993), 518–533.

# Compact Voronoi Diagrams for Moving Convex Polygons<sup>\*</sup>

Leonidas J. Guibas<sup>1</sup>, Jack Snoeyink<sup>2</sup>, and Li Zhang<sup>1</sup>

<sup>1</sup> Department of Computer Science, Stanford University, Stanford, CA 94305, USA.

<sup>2</sup> Department of Computer Science, University of North Carolina, Chapel Hill,  
NC 27599, USA.

**Abstract.** We describe a kinetic data structure for maintaining a compact Voronoi-like diagram of convex polygons moving around in the plane. We use a compact diagram for the polygons, dual to the Voronoi, first presented in [MKS96]. A key feature of this diagram is that its size is only a function of the number of polygons and not of their complexity. We demonstrate a local certifying property of that diagram, akin to that of Delaunay triangulations of points. We then obtain a method for maintaining this diagram that is output-sensitive and costs  $O(\log n)$  per update. Furthermore, we show that for a set of  $k$  polygons with a total of  $n$  vertices moving along bounded degree algebraic motions, this dual diagram, and thus their compact Voronoi diagram, changes combinatorially  $\Omega(n^2)$  and  $O(kn^2\beta(k)\beta(n))$  times, where  $\beta(\cdot)$  is an extremely slowly growing function. This compact Voronoi diagram can be used for collision detection or retraction motion planning among the moving polygons.

## 1 Introduction

Voronoi diagrams are fundamental data structures in computational geometry and have been used in a wide variety of applications that require proximity information among geometric objects. When a Voronoi diagram is defined on objects that are not points, all features of these objects can contribute to the diagram's complexity. In this paper we will be concerned with the Voronoi diagram of  $k$  convex polygons in the plane with a total of  $n$  vertices. Even though such a diagram defines only  $k$  regions (one per object), its total geometric complexity is  $\Theta(n)$  — as all polygon vertices can contribute to the linear or parabolic bisector segments defining the edges separating these regions. Many of the queries we may want to use such a diagram for, however, (such as reporting the object closest to a query point, or the closest pair among the given objects) refer only to the objects themselves and not their individual features. In 1996, McAllister, Kirkpatrick, and Snoeyink [MKS96] showed how to compute what they called

\* Leo Guibas and Li Zhang were partially supported by NSF grant CCR-9623851 and by US Army MURI grant 5-23542-A. Jack Snoeyink was formerly at the University of British Columbia, where this work was partially supported by a research grant from NSERC and the Institute of Robotics and Intelligent Systems.

a *compact* Voronoi diagram, which is a simplified partition of the plane of size  $\Theta(k)$ , but which can still be used to answer proximity queries (such as the above) about the objects efficiently.

Voronoi-based methods have been successfully used to address proximity queries in robotics applications, such as collision detection [LC91] or retraction motion planning [Lat91]. In the robotics setting the convex polygons represent obstacles to be avoided. When these obstacles move, we need to update their Voronoi diagram accordingly. A natural framework for studying this is the *kinetic data structures* (KDS) framework, introduced by Basch, Guibas, and Hershberger in [BGH97]. In the KDS setting one maintains a geometric structure under continuous motion of its defining elements through a set of certificates proving its correctness. An event queue is maintained for the failure times of these certificates and at each event the structure of interest, and its kinetic proof, are appropriately updated. It turns out that maintaining kinetically the Voronoi diagram of moving points is easy [GMR92], as a set of local conditions (`InCircle` tests) certify the global correctness of the structure and local repairs are always possible.

In this paper we study the kinetic maintenance of the compact Voronoi diagram for disjoint moving convex polygons in the plane. Though this diagram contains much less detailed information than the full Voronoi diagram, it turns out that it still has enough structure that it can be certified through a set of topologically local certificates and thus maintained as the objects move. This kinetic diagram can, for example, be used to track the closest pair among the moving objects and therefore perform collision detection. Many collision detection algorithms for convex bodies rely on determining the closest pair of features between two objects as the basic building block [LC91,Mir97]. To avoid considering all  $\binom{k}{2}$  pairs of objects, these algorithms invoke a so-called ‘broad-phase’ method to select which pairs of objects to test — usually an intersection test on bounding boxes for the objects. The compact Voronoi diagram elegantly solves the broad-phase problem and always provides us with a set of  $O(k)$  pairs of objects (those whose regions are adjacent in the diagram) that is guaranteed to contain the closest pair. As another example, the diagram can be used to solve the retraction motion planning problem, with the help of an additional structure that maintains the closest pair of features between two moving convex chains.

We introduce the basic notations and definitions we need in Section 2. A key notion is that of *junction triangles*, which are dual to the degree-3 vertices of the Voronoi diagram. If we remove the junction triangles from the free space around the obstacles, the rest of the free space can be decomposed into a set of corridors, each between two of the convex objects. This structure and the certification of its correctness are introduced in Section 3. In Section 4 we study the number of changes to the compact Voronoi diagram when the defining polygons move pseudo-algebraically in the plane. Using lower envelope and other techniques, we can show that the number of changes to the diagram is roughly  $O(kn^2)$ . Finally in Section 5 we give applications to the collision detection and retraction motion planning.

## 2 Preliminaries

A distance function  $\delta$  defined on points in  $\mathbb{R}^2$  can be generalized to points sets  $S_1$  and  $S_2$  by setting  $\delta(S_1, S_2) = \inf_{s_1 \in S_1, s_2 \in S_2} \delta(s_1, s_2)$ ; when  $S_1$  contains only one point  $s$ , we can simply write  $\delta(s, S_2)$ . If  $S_1, S_2$  are bounded closed sets, their distance can be realized by a pair of points  $(s_1, s_2)$  where  $s_1 \in S_1$  and  $s_2 \in S_2$ . In what follows we will use  $\delta$  to denote the usual Euclidean distance.

Consider now a set  $\mathcal{P}$  of disjoint convex obstacles in the plane. Under the above distance function between points and points sets, we can define a Voronoi diagram for  $\mathcal{P}$ , called the *generalized Voronoi diagram* of  $\mathcal{P}$ , which is the partition of the free space in the plane according to the nearest obstacle. We assume that  $\mathcal{P}$  is a set of  $k$  disjoint convex polygons with  $n$  vertices in total. The generalized Voronoi diagram of  $\mathcal{P}$ , denoted by  $\mathcal{V}(\mathcal{P})$ , has complexity  $O(n)$  and can be built in time  $O(n \log n)$ . In [MKS96], a compact representation of  $\mathcal{V}(\mathcal{P})$  is presented. The proposed compact Voronoi diagram has size  $O(k)$  and can be computed in  $O(k \log n)$  time, assuming each object is represented by the sorted list of its vertices in clockwise (or counterclockwise) order. Despite its compactness, this new diagram is as powerful as the generalized Voronoi diagram with regards to nearest-neighbor and other queries. Here, we will show how to maintain this compact Voronoi diagram when the convex obstacles are in motion.

From this point on, when we refer to “an obstacle,” we mean a closed convex polygon. An *edge* on a polygon is an open line segment; a *feature* of a polygon is a vertex or an (open) edge; the *size* of a polygon is the number of vertices on it. Two features are *adjacent* if their closures intersect. Three polygons are *collinear* if there is a line tangent to them simultaneously. Four polygons are *cocircular* if there is a circle tangent to them simultaneously, where a line is *tangent* to an object if it intersects the object only at its boundary, and a circle is *tangent* to an object if it intersects the object only at its boundary and its interior is disjoint from the object. Normally in computational geometry we assume that objects are in general position, and specifically in our setting that no two polygon edges are parallel, no three objects are collinear, and no four objects are cocircular. However, when objects can move, it is no longer legitimate to make the above assumption. Actually, interesting events happen exactly at the time when such a degeneracy occurs. For moving objects, by general position we mean that the above events happen at distinct discrete times (never two at once).

For any point  $p$  outside a polygon  $P$ , there is a unique point  $q$  on  $P$  realizing the distance  $\delta(p, P)$ . Equivalently, there is a unique circle that is centered at  $p$  and tangent to  $P$ . Let us denote this circle by  $\omega(p, P)$ . The radius of  $\omega(p, P)$  clearly equals  $\delta(p, P)$ . The unique feature (edge or vertex) that contains  $q$  is called the *closest feature* to  $p$ . For two disjoint convex polygons  $P$  and  $Q$ , if they do not have parallel edges, there is a unique pair of points  $(p, q)$ , where  $p \in P$  and  $q \in Q$ , that realizes  $\delta(P, Q)$ . We denote by  $o(P, Q)$  the middle point of the line segment  $pq$ . At the point  $o(P, Q)$ , we can place a minimum circle that is tangent to both  $P$  and  $Q$ .

For the polygon set  $\mathcal{P}$ , denote the convex hull of the vertices of  $\mathcal{P}$  by  $\mathcal{C}(\mathcal{P})$ . Let  $\mathcal{F}(\mathcal{P}) = \mathcal{C}(\mathcal{P}) \setminus \mathcal{P}$  denote the free space outside the polygons but inside  $\mathcal{C}(\mathcal{P})$ .

For polygons  $P_1$  and  $P_2$  in  $\mathcal{P}$  and points  $p \in P_1$  and  $q \in P_2$ , the edge  $pq$  is called a *free edge* if it does not intersect the interior of any polygon in  $\mathcal{P}$ . Two non-intersecting free edges  $pq$  and  $p'q'$ , where points  $p, p' \in P_1$  and  $q, q' \in P_2$ , define a *corridor* with respect to  $\mathcal{P}$  if and only if the region bounded by  $pq, p'q'$ , and the convex chains  $pp'$  on  $P_1$  and  $qq'$  on  $P_2$  contain no polygons of  $\mathcal{P}$ . Sometimes we will talk about the *corridor of two polygons*  $P_1$  and  $P_2$ , which is the corridor with respect to  $\{P_1, P_2\}$  defined by their outer common tangents.

For two disjoint convex polygons  $P_1$  and  $P_2$ , the bisector between them is defined to be the locus of points that are equidistant from them. It is well known that the bisector is an unbounded Jordan curve that consists of  $O(|P_1| + |P_2|)$  line segments and parabolic arcs. For presentation convenience, we also add an orientation to the bisector. The oriented bisector  $\pi(P_1, P_2)$  (abbreviated  $\pi_{12}$ ) is the bisector with the orientation so that  $P_1$  is to the left of  $\pi_{12}$ . Since the bisector is an oriented unbounded Jordan curve, we can define a linear ordering  $\prec$  on the points on  $\pi_{12}$ . For two points  $p, q \in \pi_{12}$ , we say that  $p \prec q$  if  $p$  is encountered before  $q$  when traveling on  $\pi_{12}$  consistently with the orientation of  $\pi_{12}$  (Figure 1(a)). We can parameterize the bisector  $\pi_{12}$  as follows: for a point  $p \in \pi_{12}$ , if  $p \prec o(P_1, P_2)$ , then  $\zeta_{12}(p) = -(\delta(p, P_1) - \delta(o, P_1))$ ; otherwise,  $\zeta_{12}(p) = \delta(p, P_1) - \delta(o, P_1)$ . When there is no degeneracy, the function  $\zeta_{12}$  is an one-to-one and onto mapping from  $\pi_{12}$  to  $\mathbb{IR}$  (MKS96). Clearly,  $\zeta_{12} = -\zeta_{21}$ .

For three convex polygons  $P_1, P_2$  and  $P_3$  in general position, it is known that the bisectors  $\pi_{12}$  and  $\pi_{13}$  intersect at most twice. For an intersection  $v$  between  $\pi_{12}$  and  $\pi_{13}$ , the circle  $\omega(v, P_1)$  is tangent to  $P_1, P_2$  and  $P_3$ . We say that  $v$  is defined by the ordered triplet  $(P_1, P_2, P_3)$  if  $P_1, P_2$  and  $P_3$  are tangent to  $\omega(v, P_1)$  in counterclockwise direction. Then there is at most one vertex defined by  $(P_1, P_2, P_3)$ .

A point  $p \in \pi_{12}$  is said to be *shaded* by  $P_3$  if  $\omega(p, P_1) \cap P_3 \neq \emptyset$ . Let  $S_{12,3}$  denote the set of the points on  $\pi_{12}$  shaded by  $P_3$ . Consider the set of parameter values represented by the shaded portion,  $\{\zeta_{12}(p) \mid p \in S_{12,3}\}$ , which we denote by  $\tilde{S}_{12,3}$ . Since bisectors  $\pi_{12}$  and  $\pi_{13}$  intersect at most twice, the shaded set  $\tilde{S}_{12,3}$  must have the form  $\emptyset, (-\infty, a], [b, +\infty), (-\infty, a] \cup [b, +\infty), [a, b]$ , or  $(-\infty, +\infty)$ , where  $a, b$  correspond to the parameter values of the Voronoi vertices defined by  $P_1, P_2$  and  $P_3$ .

When  $\tilde{S}_{12,3}$  has the form  $(-\infty, a]$  or  $(-\infty, a] \cup [b, +\infty)$ , we say that  $\pi_{12}$  is *half-shaded* by  $P_3$  at  $a$ . Notice that if neither  $\pi_{12}$  nor  $\pi_{21}$  is half-shaded by  $P_3$ , then  $\tilde{S}_{12,3}$  must have the form  $\emptyset$  or  $[a, b]$  where  $a < b$ . The following fact is useful later in bounding the number of combinatorial changes.

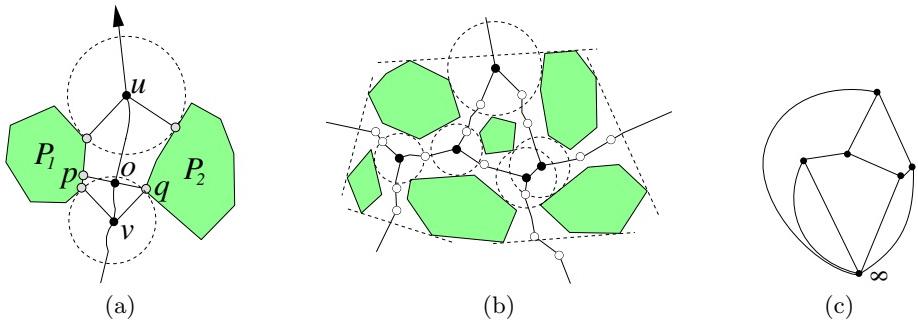
**Lemma 1.** *The shaded set  $\tilde{S}_{12,3}$  is of the form  $[a, b]$ , where  $a < b$ , if and only if  $P_3$  lies completely inside the corridor between  $P_1$  and  $P_2$ .*

The bisector  $\pi_{12}$  divides the plane into two regions that contain  $P_1$  and  $P_2$ , respectively. Let us denote  $\tau_{P_1 P_2}$  (or simply  $\tau_{12}$ ) the region that contains  $P_1$ . Each point in  $\tau_{12}$  is closer to  $P_1$  than to  $P_2$ . For  $P_i \in \mathcal{P}$ , the Voronoi region  $V(P_i)$  of  $P_i$  is then defined to be  $\bigcap_{j \neq i} \tau_{ij}$  — the set of points that are closer to  $P_i$  than to any other polygon in  $\mathcal{P}$ . Each Voronoi region is connected and bounded by portions of bisectors. Therefore, the corresponding Voronoi diagram  $\mathcal{V}(\mathcal{P})$  is a

planar map (Figure 1(b)). Two convex polygons are *adjacent* in  $\mathcal{V}(\mathcal{P})$  if their Voronoi regions share a boundary.

**Lemma 2.** *A point  $p \in \pi_{12}$  is in  $\mathcal{V}(\mathcal{P})$  if and only if  $\zeta_{12}(p)$  is not in the interior of  $\tilde{S}_{12,i}$ , for any  $i \neq 1, 2$ .*

In  $\mathcal{V}(\mathcal{P})$  there are two types of vertices, as illustrated in Figure 1(b). Vertices of degree three are *junction vertices*, corresponding to the intersections between two bisectors. The remaining degree two vertices are *interior vertices*, lying along a single bisector where the closest feature pair changes. At each junction vertex, we can grow a circle that touches three polygons and is free of other polygons. This circle is called a *witness circle* for that junction vertex.



**Fig. 1.** (a) The oriented bisector  $\pi_{12}$  between  $P_1$  and  $P_2$ . The pair  $(p, q)$  is the closest pair of points between  $P_1$  and  $P_2$ . According to the ordering,  $v \prec o \prec u$ . (b) The generalized Voronoi diagram where the solid vertices are junction vertices and the hollow ones are interior vertices. Not all hollow vertices are shown. (c) The compact Voronoi diagram with only the junction vertices remaining. A point at  $\infty$  is added to compactify the diagram.

The junction vertices capture the topology of  $\mathcal{V}(\mathcal{P})$ . Furthermore, while the total number of vertices in  $\mathcal{V}(\mathcal{P})$  can be  $\Theta(n)$ , the number of junction vertices is at most  $2k - 4$ , where  $k$  is the number of polygons in  $\mathcal{P}$ . The compact Voronoi diagram is based on these junction vertices, plus an additional imaginary vertex at  $\infty$ : we form an edge between two vertices if and only if there is a portion of a bisector between them with no junction vertex in between (Figure 1(c)). If a portion of a bisector extends to infinity, we connect the vertex to the node at  $\infty$ . We refer to this diagram as the *compact Voronoi diagram* of  $\mathcal{P}$ . In [MKS96], it is shown that the compact Voronoi diagram has many useful properties and that it can be computed in  $O(k \log n)$  time. In the following, we study how to maintain the compact Voronoi diagram for moving polygons.

### 3 Maintaining the Compact Voronoi Diagram for Moving Obstacles

Maintaining the traditional Voronoi diagram of moving points in the plane, usually represented via its dual Delaunay triangulation, is straightforward [GMR92]. This is because a local condition, the ‘empty circle’ property for triangulation edges with pairs of adjacent triangles [PS90], implies the global correctness of the diagram. When one of these conditions fails due to object motion, a local transformation (the ‘edge-flip’) is sufficient to restore local, and therefore global, correctness. The same principle was exploited to maintain the power diagrams of moving balls in [GZ98].

To pursue this idea in our current context, we first define a dual ‘triangulation’ of the (compact) Voronoi diagram. Recall that  $\mathcal{F}(\mathcal{P})$  denotes the free space inside the convex hull of  $\mathcal{P}$ . A triangle is called a *junction triangle* if it is incident to three objects in  $\mathcal{P}$ . A *corridor* is a four-sided portion of the free space delimited by two polygons on two opposite sides. With slight abuse of notations, we call a cell decomposition with triangles and corridors as primitive cells a *triangulation*. A triangulation  $\mathcal{T}(\mathcal{P})$  of  $\mathcal{P}$  is a cell decomposition of  $\mathcal{F}(\mathcal{P})$  into junction triangles and corridors with all the vertices of  $\mathcal{T}$  on polygon boundaries. that satisfy the following properties:

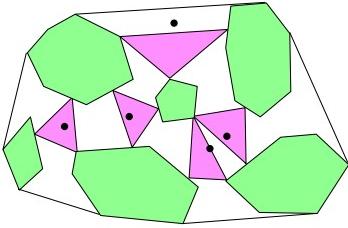
- All the (junction) triangles and corridors in  $\mathcal{T}$  are interior disjoint,
- The vertices of triangles and corridors in  $\mathcal{T}$  are on polygon boundaries,
- The triangles in  $\mathcal{T}$  do not intersect the interior of any polygon  $P \in \mathcal{P}$ , and
- The union of triangles and corridors in  $\mathcal{T}$  covers  $\mathcal{F}(\mathcal{P})$ .

For a given polygon set  $\mathcal{P}$ , the number of junction triangles in a triangulation are specified by the following fact.

**Lemma 3.** *For  $k$  disjoint convex polygons, if their convex hull contains  $h$  non-polygon edges, then any triangulation of their free space contains exactly  $2k - h - 2$  junction triangles.*

We can form a triangulation of  $\mathcal{F}(\mathcal{P})$  based on the compact Voronoi diagram of  $\mathcal{P}$  as follows. For each junction vertex in the Voronoi diagram, its witness circle touches three polygons. We connect the three contact points to form a junction triangle corresponding to each junction vertex. It is easy to prove that the junction triangles thus formed do not intersect polygon interior and do not interpenetrate each other.

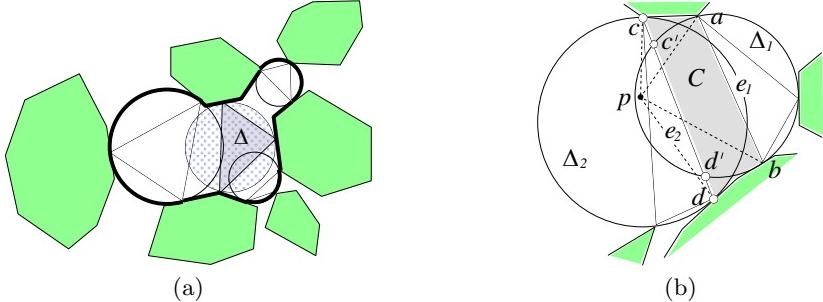
If we remove these junction triangles from  $\mathcal{F}(\mathcal{P})$ , we will have a set of disconnected regions where each connected component is a corridor between two convex polygons. The corridors and junction triangles together form a triangulation that we will call the *Delaunay triangulation* of  $\mathcal{P}$  and denote it by  $\mathcal{D}(\mathcal{P})$  (Figure 2). If a junction triangle  $\Delta$  is incident to three features  $f_1, f_2$  and  $f_3$ , on different polygons, we say that the triplet  $(f_1, f_2, f_3)$  defines  $\Delta$ . When objects start to move, the Delaunay triangulation changes combinatorially if such a triplet changes. In the following, we will show how to maintain  $\mathcal{D}(\mathcal{P})$  when the objects in  $\mathcal{P}$  move. We assume that during the motion, all the objects remain disjoint. As we will see later, we can use  $\mathcal{D}(\mathcal{P})$  to detect collisions.



**Fig. 2.** The Delaunay triangulation of the compact Voronoi diagram. The free space between two adjacent junction triangles are corridors.

What is crucial for the maintenance of  $\mathcal{D}(\mathcal{P})$  is its local certification property. For any junction triangle  $\Delta$ , if  $\Delta$ 's circumcircle (denoted by  $\tilde{\Delta}$ ) does not intersect the interior of any  $P \in \mathcal{P}$ ,  $\Delta$  is called a *Delaunay* triangle. Clearly, a triangulation is a Delaunay triangulation if and only if all of its junction triangles are Delaunay triangles. To state the local property we seek, we define a local Delaunay condition for a junction triangle. A corridor is said to be *adjacent* to a junction triangle if they share an edge. Two junction triangles are *adjacent* if they are adjacent to the same corridor. Each junction triangle  $\Delta$  then has up to three adjacent corridors and three adjacent triangles (Figure 3 (a)). The neighboring polygons for  $\Delta$  are defined to be the polygons incident to it or to one of its adjacent junction triangles. We say that a junction triangle is *locally Delaunay* if its circumcircle does not intersect any of its neighboring polygons. Clearly, all the junction triangles of  $\mathcal{D}(\mathcal{P})$  are locally Delaunay.

angle  $\Delta$  then has up to three adjacent corridors and three adjacent triangles (Figure 3 (a)). The neighboring polygons for  $\Delta$  are defined to be the polygons incident to it or to one of its adjacent junction triangles. We say that a junction triangle is *locally Delaunay* if its circumcircle does not intersect any of its neighboring polygons. Clearly, all the junction triangles of  $\mathcal{D}(\mathcal{P})$  are locally Delaunay.



**Fig. 3.** (a) A junction triangle and its adjacent corridors and polygons. When  $\Delta$  is locally Delaunay,  $\tilde{\Delta}$  is covered by the adjacent corridors and circumcircles of its adjacent junction triangles, where the boundary of this region is thickened in the figure. (b) Proof of the local property.

We will show that the other direction also holds, which is the counterpart of the local certification property of the Delaunay triangulation for points.

**Lemma 4 (Local Property).** *If all the junction triangles of a triangulation  $\mathcal{T}(\mathcal{P})$  are locally Delaunay, then  $\mathcal{T}(\mathcal{P})$  is the Delaunay triangulation of  $\mathcal{P}$ .*

*Proof.* First, we observe that if a triangle  $\Delta$  is locally Delaunay, then  $\tilde{\Delta}$  is covered by the union of  $\Delta$ ,  $\Delta$ 's adjacent corridors, and the circumcircles of  $\Delta$ 's adjacent triangles (Figure 3 (a)).

Using the above fact, we will prove the local property by contradiction. Assume that  $\mathcal{T}$  is not a Delaunay triangulation — then there must exist a point

$p$  interior to one of the polygons in  $\mathcal{P}$  and a junction triangle  $\Delta_1 \in T$  so that  $p \in \tilde{\Delta}_1$ . For an edge  $e_1$  of  $\Delta_1$ , denote by  $\tilde{e}$  the sector of  $\tilde{\Delta}_1$  bounded by the chord  $e_1$ . By the definition of triangulation,  $\Delta_1$  does not intersect any polygon interior. Therefore,  $p \in \tilde{\Delta}_1 \setminus \Delta_1$ , i.e.,  $p \in \tilde{ab}$  for a unique edge  $ab$  of  $\Delta_1$ . Denote the angle  $apb$  by  $\theta(p, \Delta_1)$ . We note that  $\theta(p, \Delta_1)$  is the maximum angle  $p$  can make with the endpoints of edges of  $\Delta_1$ . Among all the junction triangles whose circumcircles contain  $p$ , let  $\Delta_1$  be the one with the maximum  $\theta(p, \Delta_1)$  and let  $e_1$  be the edge of  $\Delta_1$  so that  $p \in \tilde{e}_1$ .

Now consider the corridor  $C$  adjacent to  $\Delta_1$  at  $e_1$ . Let the other edge bounding  $C$  to be  $e_2$  and the junction triangle incident to  $e_2$  be  $\Delta_2$ . Since the corridor is the union of a set of triangles in  $\mathcal{T}$ , it cannot contain  $p$ . Thus,  $p \in \tilde{e}_1 \setminus C$ . This implies that  $\tilde{e}_1$  intersects the edge  $e_2$  since  $\tilde{\Delta}_1$  does not intersect the polygons incident to  $\Delta_1$ . By the local Delaunay property of  $\Delta_1$ ,  $\tilde{\Delta}_1$  is covered by the union of corridors and the circumcircles of adjacent triangles, i.e.,  $\tilde{e}_1 \subset \tilde{\Delta}_2 \cup C$ , or  $\tilde{e}_1 \setminus C \subset \tilde{\Delta}_2$ . Therefore,  $p \in \tilde{\Delta}_2$ . We can further conclude that  $p$  is not in  $\tilde{e}_2$  because  $\tilde{e}_1 \cap \tilde{e}_2 \subset C$ . Suppose that the endpoints of  $e_2$  are  $c$  and  $d$  and that  $e_2$  intersects the boundary of  $\tilde{\Delta}_1$  at points  $c'$  and  $d'$ . Then clearly the angle  $\theta(p, \Delta_2) > \angle cpd > \angle c'pd' > \angle apb = \theta(p, \Delta_1)$ , contradicting the maximality of  $\theta(p, \Delta_1)$  (Figure 3 (b)).

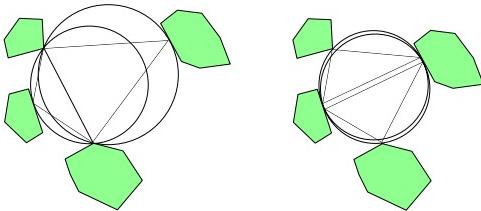
By the above local property, to certify that a triangulation is the Delaunay triangulation, it is sufficient to certify that all the junction triangles are locally Delaunay, i.e., the circumcircle of each junction triangle does not intersect the interior of any of its neighboring polygons. Among the (up to six) neighboring polygons of a junction triangle  $\Delta$ , three are incident to  $\Delta$  and the others are incident to one of the adjacent junction triangles to  $\Delta$ . We consider these two cases separately. In the following, we assume that  $\Delta$  is defined by the triplet of features  $(f_1, f_2, f_3)$  where  $f_1, f_2, f_3$  are features on  $P_1, P_2$  and  $P_3$ , respectively.

To certify that  $\tilde{\Delta}$  does not intersect the interior of its incident polygons it suffices, by convexity, to certify that  $\tilde{\Delta}$  does not intersect any features adjacent to  $f_1, f_2$  or  $f_3$ . All these certificates can be written as algebraic conditions in terms of the coordinates of (constant number of) polygon vertices. When a certificate fails, say, when  $\tilde{\Delta}$  intersects the feature  $f'_1$ , an adjacent feature of  $f_1$ , we then simply update the triplet of features from  $(f_1, f_2, f_3)$  to  $(f'_1, f_2, f_3)$ . The topological structure of  $\mathcal{D}(\mathcal{P})$  is not affected by such events.

For those neighboring polygons that are not incident to  $\Delta$ , suppose that  $\Delta_1$  is a junction triangle adjacent to  $\Delta$  and it is incident to  $P_1, P_2$  and  $P_4$ . To certify that  $\tilde{\Delta}$  does not intersect  $P_4$ , it suffices to certify that  $\tilde{\Delta}$  does not contain the point that is on  $\Delta_1$  on  $P_4$  because  $\tilde{\Delta}_1$  is disjoint from the interior of  $P_1, P_2$  and  $P_4$ . When such a certificate fails, it must be the case that the circumcircles of  $\Delta$  and  $\Delta_1$  are coincident. In other words, this happens when  $P_1, P_2, P_3, P_4$  are cocircular. We call such events *cocircularity events*. We also note the special case when the junction triangle is on the convex hull. In this case, the corresponding event is when the polygons become collinear and the common tangent line supports the convex hull of  $\mathcal{P}$  or vice versa. Such event decreases or increases the number of junction triangles by one. For such *collinearity events*,

we can watch each junction triangle with an edge on the boundary of  $\mathcal{C}(\mathcal{P})$  to see when such a triangle degenerates and each pair of adjacent non-polygonal convex hull edges to see when such a triangle emerges. In the following, we will focus on the cocircularity events.

For  $\Delta, \Delta_1$  to have the same circumcircle, they must share an edge, say  $e_1$ , which is incident to  $P_1$  and  $P_2$  — otherwise, we would contradict the fact that  $\tilde{\Delta}$  and  $\tilde{\Delta}_1$  do not intersect the interior of  $P_1$  and  $P_2$ . After such a cocircularity event happens, the triangulation is no longer a valid Delaunay triangulation. To fix it, we simply delete the edge  $e_1$  and add the other diagonal of the quadrangle formed by the union  $\Delta \cup \Delta_1$  (Figure 4).



**Fig. 4.** The cocircularity event and edge-flip operation.

appears or disappears. The other types of events do not affect the number of junction triangles. Therefore, we also maintain the right number of junction triangles. By Lemma 3 and the local property, we can conclude that the above algorithm correctly maintains  $\mathcal{D}(\mathcal{P})$ , and thus the compact Voronoi diagram.

We have shown above how the compact Voronoi diagram can be maintained. Since the number of certificates is the sum of the number of junction triangles and the non-polygonal convex hull edges, the above structure has  $O(k)$  certificates. The processing time for each event is  $O(\log k)$ , dominated by the processing cost of the event queue. However, the structure is not local as one polygon may be involved in up to  $\Theta(k)$  certificates, although on the average each polygon is involved in  $O(1)$  certificates. As for the events processed, the above algorithm is output-sensitive in the sense that every event changes either the features that define a junction Voronoi vertex or the topology of the compact Voronoi diagram. Thus, we have,

**Theorem 1.** *The compact Voronoi diagram of  $\mathcal{P}$  can be maintained by a kinetic data structure in an output-sensitive manner. In the structure, the number of certificates is  $O(k)$ , and each event can be processed in  $O(\log k)$  time.*

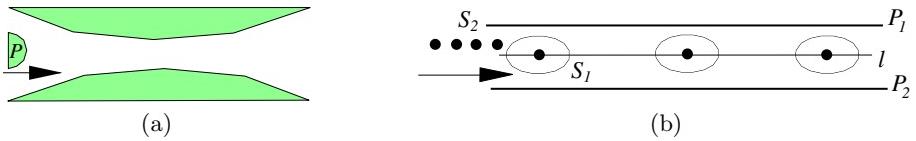
According to [BGH97], the structure we described is compact, responsive, and efficient. However, it is not local as one polygon may be involved up to  $\Theta(k)$  certificates. Although the algorithm maintains the compact Voronoi diagram in an output-sensitive manner, we have not answered the question on how many events the data structure may need to process for algebraic polygon motions. To complete our analysis, in the next section, we will analyze the number of the combinatorial changes of the compact Voronoi diagram.

We shall see that the above algorithm maintains a set of junction triangles that satisfy local Delaunay property. Further, note that only the collinearity events may change the number of junction triangles — we decrease or increase the number of junction triangles by one, depending on whether a convex hull edge

## 4 Combinatorial Changes of the Compact Voronoi Diagram

In this section, we will study the number of combinatorial changes of the compact Voronoi diagram. For the analysis purpose, we assume that the polygons move rigidly in pseudo-algebraic motion with constant degree. That is, each certificate involving the same set of features can fail only constant number of times during the entire motion process. This assumption is very general as it includes the motions that can be represented in algebraic or rational functions with constant degree.

Observe that an event happens when there are four features cocircular or three collinear. This gives us an immediate upper bound of  $O(n^4)$  for constant degree pseudo-algebraic motion. However, we will show a significantly smaller upper bound of  $O(kn^2\beta(k)\beta(n))$ . Here,  $\lambda(n)$  is the maximum length of a  $(n, s)$  Davenport-Schinzel sequence for some constant  $s$ , and  $\beta(n) = \lambda(n)/n$  is an extremely slowly growing function and can be regarded as close to a constant for all reasonable values of  $n$ . On the other hand, there are examples to show an  $\Omega(n^2)$  lower bound. The lower bound can be realized by three convex polygons, each with  $n/3$  edges. In Figure 5(a), when the polygon  $P$  moves horizontally, the triplets of features that define the junction Voronoi vertices change  $\Omega(n^2)$  times.



**Fig. 5.** Lower bound constructions: (a) shows an example with  $\Omega(n^2)$  changes to the compact Voronoi diagram. (b) shows an example with  $\Omega(k^2)$  changes to a pair of objects.

Now, our main task is to prove the following theorem.

**Theorem 2.** *Suppose that  $\mathcal{P}$  is a set of  $k$  disjoint convex polygons with  $n$  vertices in total. When all the polygons in  $\mathcal{P}$  move algebraically and without colliding with each other, the compact Voronoi diagram changes  $O(n\lambda(n)\lambda(k)) = O(kn^2\beta(n)\beta(k))$  times.*

In [HKK92], a similar upper bound is proved for the number of changes of the Voronoi diagram of  $k$  sets of points, each moving independently and rigidly. We will use similar techniques to prove our upper bound. However, the presence of edges adds complexity, and additional insights are required to complete the proof.

We first give an example to show the difference from the points case. In the proof of an  $O(k^3\beta(k))$  bound on the number of changes of the Voronoi diagram

of  $k$  moving points, the key step is to bound the number of changes involving two points by  $O(k\beta(k))$ ; this implies that the total number of changes is bounded by  $O(k^3\beta(k))$ . This fact, however, is no longer true for polygons. Consider the example shown in Figure 5 (b), where  $P_1$  and  $P_2$  are two long, parallel line segments and  $S_1$  and  $S_2$  are two group of points. Let  $\ell$  be the bisector line of  $P_1$  and  $P_2$ . The set  $S_1$  consists of  $k$  points lying on  $\ell$  so that their Voronoi regions relative to  $P_1$  and  $P_2$  are mutually disjoint. The other group of points  $S_2$  lie slightly above  $\ell$  and are spaced compactly so that they, together with their Voronoi regions, can be accommodated between any two points of  $S_1$ . Now, imagine the second group of points moving from  $x = -\infty$  to  $x = +\infty$  on the line  $\ell$ ; there will be  $\Theta(k^2)$  changes of the Voronoi edges incident to  $P_1$  and  $P_2$ . As this example shows, we can no longer bound the number of changes related to a pair of polygons by roughly  $O(k)$  as was done in the point case. However, as we shall see later, still there is a way to charge each change to a carefully chosen polygon pair so that no pair receives too many charges.

To proceed, we first consider how Voronoi vertices move when polygons move. Consider three convex polygons, i.e.,  $\mathcal{P} = \{P_1, P_2, P_3\}$ . We write that  $n_i = |P_i|$ , where  $1 \leq i \leq 3$ , and  $n = n_1 + n_2 + n_3$ . For three objects, a combinatorial change can happen to  $\mathcal{D}(\mathcal{P})$  only when the triplets of features that define the Voronoi vertices change. For the number of combinatorial changes for constant degree algebraic motion, we have the following bound.

**Lemma 5.** *The compact Voronoi diagram of three convex polygons changes  $O((n_1n_2 + n_1n_3 + n_2n_3)\beta(n))$  times.*

*Proof.* There are two types of events causing a combinatorial change.

1. The *first* type (type I) occurs when there is a common outer tangent line to  $P_1$ ,  $P_2$  and  $P_3$ . Such events cause a Voronoi vertex to appear or disappear.
2. The *second* type (type II) occurs when there is a circle touching four features, i.e., when four features become *cocircular*, and this circle is free of the interior of the polygons. Such events change the triplet of features that define a particular Voronoi vertex.

It is easy to see that there can be at most  $O(n_1n_2 + n_1n_3 + n_2n_3)$  type I events. For the type II events, among the four features that are cocircular, there must be two, say  $f_1$  and  $f_2$ , from the same polygon. By the convexity, they are an edge and an endpoint of the edge. Therefore, when this event happens, the circle must be tangent to an edge at its endpoint. Suppose that  $e$  is an edge of  $P_1$  and  $p$  is an endpoint of  $e$ . We parameterize all the circles tangent to  $e$  at  $p$  by their radii (we only consider those circles whose centers are outside of the polygon), and denote this set by  $\mathcal{O}$ . Then, for any feature  $f$ , we define a function  $\delta_f(t)$  to be the radius of the circle in  $\mathcal{O}$  which touches  $f$  at time  $t$ . When  $f$  moves algebraically,  $\delta_f(t)$  is a rational function. Consider the family of function  $\Xi(P_2) = \{\delta_f(t) \mid f \text{ is a feature of } P_2\}$ . Clearly, the lower envelope of the arrangement of  $\Xi(P_2)$  corresponds to the circles that touch  $P_2$ . Similarly, we can define  $\Xi(P_3)$ . Then, a type II event that involves  $e$  and  $p$  corresponds to an intersection point between the lower envelopes of  $\Xi(P_2)$  and  $\Xi(P_3)$ , whose

complexity is bounded by  $\lambda(n_2) + \lambda(n_3)$ . Therefore, the number of type II events that involve two features of  $P_1$  is bounded by  $O(n_1(\lambda(n_2) + \lambda(n_3))) = O((n_1 n_2 + n_1 n_3)\beta(n))$ . Repeating the above argument for  $P_2$  and  $P_3$  proves the lemma.

Because of the way that we parameterize the bisector, we also need the following fact.

**Lemma 6.** *The distance function  $\delta(P_1, P_2)$  consists of  $O(n_1 n_2)$  rational arcs with constant degree when  $P_1, P_2$  moves algebraically with constant degree.*

For three polygons  $P_1$ ,  $P_2$  and  $P_3$ , recall that  $S_{12,3}$  is the set of points that are on  $\pi_{12}$  and shaded by  $P_3$ . The parameters of this shaded set,  $\tilde{S}_{12,3}$ , may be of the forms  $\emptyset$ ,  $(\infty, a]$ ,  $[b, +\infty)$ ,  $(-\infty, a] \cup [b, +\infty)$ ,  $[a, b]$ , or  $(-\infty, +\infty)$ . We define the function  $\phi_{12,3}(t)$  as follows. If at time  $t$ ,  $\pi_{12}$  is half-shaded by  $P_3$  at  $a$ , then  $\phi_{12,3}(t)$  is defined to be  $a$ . Otherwise, it is undefined. Since an endpoint of  $\tilde{S}_{12,3}$  corresponds to the parameter value of a Voronoi vertex when considering  $P_1$ ,  $P_2$  and  $P_3$  only, Lemma 5 and 6 say that  $\phi_{12,3}$  consists of  $O((n_1 n_2 + n_1 n_3 + n_2 n_3)\beta(n))$  pieces of rational arcs. Likely, we define the function  $\phi_{ij,l}$  for each triplet  $i, j, l$ .

For a pair of polygons  $P_i$  and  $P_j$ , we have a family of functions  $\Phi_{ij} = \{\phi_{ij,l} \mid l \neq i, j\}$ . Let  $\Gamma(\Phi)$  denote the upper envelope of a set of functions  $\Phi$ . We first show that

**Lemma 7.** *Each cocircular event can be charged to a break point on  $\Gamma(\Phi_{ij})$  or the overlay between  $\Gamma(\Phi_{ij})$  and  $-\Gamma(\Phi_{ji})$ , for some  $i \neq j$ .*

For the moment, let us assume that the above lemma is true and prove Theorem 2.

*Proof.* (**Theorem 2**). As we have already discussed, there are two types of events: type I, when the feature triplets defining Voronoi vertices change, and type II, when four polygons become cocircular. By Lemma 5, the number of type I of events is bounded by:

$$\sum_{i,j,l} (n_i n_j + n_i n_l + n_j n_l) \beta(n) = O(kn^2 \beta(n)).$$

For type II events, by Lemma 7, they can be charged to break points on the lower or upper envelopes of  $\Phi_{ij}$ 's or their overlay. Since each  $\phi_{ij,l}$  consists of  $O((n_i n_j + n_i n_l + n_j n_l)\beta(n))$  pieces of rational arcs. The complexity of  $\Gamma(\Phi_{ij})$  is then bounded by:

$$\beta(k) \sum_l (n_i n_j + (n_i + n_j) n_l) \beta(n) = O((kn_i n_j + (n_i + n_j) n) \beta(n) \beta(k)).$$

The overlay between two envelopes has the same order of complexity. Thus, the number of cocircular events is bounded by

$$\sum_{i,j} (O((kn_i n_j + (n_i + n_j) n) \beta(n) \beta(k))) = O(kn^2 \beta(n) \beta(k)).$$

Now, the only piece left is the proof of Lemma 7.

*Proof.* (**Lemma 7**). Suppose that at time  $t$ , a cocircularity event happens to  $P_1$ ,  $P_2$ ,  $P_3$  and  $P_4$ . We claim that among those four polygons, there always exist two, say  $P_1$  and  $P_2$ , so that  $\tilde{S}_{12,3}$  and  $\tilde{S}_{12,4}$  are not closed intervals with the form

$[a, b]$ . Now pick any two polygons, say  $P_1$  and  $P_2$ . If neither of the shaded sets  $S_{12,3}$  nor  $S_{12,4}$  is a bounded arc, then the pair of polygons  $P_1, P_2$  are what we want. Otherwise, suppose  $S_{12,3}$  is bounded. By Lemma 1,  $P_3$  is completely inside the corridor between  $P_1$  and  $P_2$ . Thus  $P_2$  is not inside the corridor between  $P_1$  and  $P_3$ . If  $P_4$  is not inside the corridor between  $P_1$  and  $P_3$ , then the pair  $P_1, P_3$  satisfy the requirement. Otherwise,  $P_4$  is inside the corridor between  $P_1$  and  $P_3$ . In this case,  $P_3, P_4$  are the desired pair. Let us rename the pair with the above property  $P_1, P_2$ .

Suppose  $v$  is the coincident Voronoi vertex at time  $t$ . Let  $x = \zeta_{12}(v)$ . By Lemma 2, at time  $t$ , there cannot be any other  $P_i$  ( $i \neq 1, 2, 3, 4$ ) so that  $x$  in the interior of  $\tilde{S}_{12,i}$ . Since  $\tilde{S}_{12,3}$  are not closed intervals, either  $\phi_{12,3}(t) = x$  or  $\phi_{21,3}(t) = -x$ . The same argument applies to  $\phi_{12,4}(t)$  and  $\phi_{21,4}(t)$ . The fact that  $v$  is not shaded by any other  $P_i$  allows us to charge such an event either to a break point on  $\Gamma(\Phi_{12})$  or  $\Gamma(\Phi_{21})$  or to an intersection between  $\Gamma(\Phi_{12})$  and  $-\Gamma(\Phi_{21})$ . Thus we have proved Lemma 7 and completed the proof of Theorem 2.

## 5 Applications

In this section, we briefly discuss some applications of the above data structure. In the above presentation, one important issue left unspecified in our method is the way to handle the corridors. We will present different structures dependent on the application requirements.

### 5.1 Collision Detection

A major motivation to maintaining a decomposition of free space is to detect collision for moving objects [EGSZ99, BEG<sup>+</sup>99]. If for each corridor, we add an inner bi-tangent line between the two convex chains bounding the corridor, we can detect collision between the objects involved (refer to [EGSZ99] for why we prefer tangent based separation to the separation based on the closet pair). In [EGSZ99], efficient hierarchical methods are developed to reduce the number of events associated with tracking tangents. Those methods can be used in our setting as well.

### 5.2 Retraction Motion Planning

The compact Voronoi diagram can be used to do retraction motion planning (so that the robot finds a path that stays maximally far from the obstacles). In retraction motion planning, we need to know the narrowest passage between two convex polygons. For this purpose, we may maintain the closest pair of features between two convex chains in a corridor. In [LC91, Mir97], there are local conditions given to check if a pair of features is the closest pair. It is not hard to see that such a condition can be used to certify and then maintain the nearest pair.

## 6 Conclusion

We have shown how to maintain a partition of the free space outside  $k$  moving convex polygons in the plane into triangles and corridors. In each cell of this partition the closest obstacle is one of the two or three polygons defining the corridor or triangle respectively. Our structure continuously maintains  $O(k)$  polygon pairs among which must be the closest pair of polygons. With the addition of a simple corridor collision test, as outlined above, the kinetic compact Voronoi diagram subsumes both the broad and narrow phases as commonly defined in the collision detection literature. Unlike more classical methods, our structure can easily accommodate deforming obstacles, as long as they stay convex. An extension of our structure to 3D would be interesting.

## References

- BEG<sup>+</sup>99. J. Basch, J. Erickson, L. J. Guibas, J. Hershberger, and L. Zhang. Kinetic collision detection for two simple polygons. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, pages 102–111, 1999.
- BGH97. J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 747–756, 1997.
- EGSZ99. Jeff Erickson, L. J. Guibas, Jorge Stolfi, and L. Zhang. Separation-sensitive kinetic collision detection for convex objects. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, 1999.
- GMR92. L. J. Guibas, J. S. B. Mitchell, and T. Roos. Voronoi diagrams of moving points in the plane. In G. Schmidt and R. Berghammer, editors, *Proc. 17th Internat. Workshop Graph-Theoret. Concepts Comput. Sci.*, volume 570 of *Lecture Notes Comput. Sci.*, pages 113–125. Springer-Verlag, 1992.
- GZ98. L. Guibas and L. Zhang. Euclidean proximity and power diagram. In *Proc. 10th Canadian Conference on Computational Geometry*, 1998.
- HKK92. D. P. Huttenlocher, K. Kedem, and J. M. Kleinberg. On dynamic Voronoi diagrams and the minimum Hausdorff distance for point sets under Euclidean motion in the plane. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 110–119, 1992.
- Lat91. J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, 1991.
- LC91. M. C. Lin and J. F. Canny. Efficient algorithms for incremental distance computation. In *Proc. IEEE Internat. Conf. Robot. Autom.*, volume 2, pages 1008–1014, 1991.
- Mir97. B. Mirtich. V-Clip: fast and robust polyhedral collision detection. Technical Report TR-97-05, Mitsubishi Electrical Research Laboratory, 1997.
- MKS96. M. McAllister, D. Kirkpatrick, and J. Snoeyink. A compact piecewise-linear Voronoi diagram for convex sites in the plane. *Discrete Comput. Geom.*, 15:73–105, 1996.
- PS90. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 3rd edition, October 1990.

# Efficient Expected-Case Algorithms for Planar Point Location

Sunil Arya<sup>1</sup>, Siu-Wing Cheng<sup>\*,1</sup>, David M. Mount<sup>\*\*,2</sup>, and H. Ramesh<sup>3</sup>

<sup>1</sup> Department of Computer Science,  
The Hong Kong University of Science and Technology,  
Clear Water Bay, Kowloon, Hong Kong  
`{arya, scheng}@cs.ust.hk`

<sup>2</sup> Department of Computer Science and Institute for Advanced Computer Studies,  
University of Maryland, College Park, Maryland  
`mount@cs.umd.edu`

<sup>3</sup> Department of Computer Science and Automation, Indian Institute of Science,  
Bangalore, India  
`ramesh@csa.iisc.ernet.in`

**Abstract.** Planar point location is among the most fundamental search problems in computational geometry. Although this problem has been heavily studied from the perspective of worst-case query time, there has been surprisingly little theoretical work on expected-case query time.

We are given an  $n$ -vertex planar polygonal subdivision  $S$  satisfying some weak assumptions (satisfied, for example, by all convex subdivisions). We are to preprocess this into a data structure so that queries can be answered efficiently. We assume that the two coordinates of each query point are generated independently by a probability distribution also satisfying some weak assumptions (satisfied, for example, by the uniform distribution).

In the decision tree model of computation, it is well-known from information theory that a lower bound on the expected number of comparisons is  $\text{entropy}(S)$ . We provide two data structures, one of size  $O(n^2)$  that can answer queries in  $2 \text{entropy}(S) + O(1)$  expected number of comparisons, and another of size  $O(n)$  that can answer queries in  $(4 + O(1/\sqrt{\log n})) \text{entropy}(S) + O(1)$  expected number of comparisons. These structures can be built in  $O(n^2)$  and  $O(n \log n)$  time respectively. Our results are based on a recent result due to Arya and Fu, which bounds the entropy of overlaid subdivisions.

## 1 Introduction

Planar point location is certainly among the most fundamental search problems in computational geometry. Given a polygonal subdivision  $S$  in the plane, the problem is to construct a data structure so that given any query point  $q$  in the

---

\* Research supported in part by RGC Grant HKUST 6088/99E.

\*\* Research supported in part by NSF Grant CCR-9712379.

plane, it is possible to determine efficiently which polygon of the subdivision contains  $q$ . This problem has been heavily studied in computational geometry. (For example, a search for “point location” found 77 papers in the computational geometry bibliography.) With only a few exceptions, previous work on this problem has dealt with the worst-case complexity of this problem. When expected-case complexity has been considered, it has been done under the assumption that both the subdivision and the query points are selected subject to various assumptions on distribution. Here, we consider search algorithms that are efficient in the expected-case for queries, and in the worst-case for subdivisions.

The planar point location problem is a generalization of the well-known one-dimensional search problem. In the one-dimensional case, we are given a set of  $n$  keys, and told the probabilities of accessing each key and the  $n + 1$  failure probabilities of falling in the gaps between the keys. If we assume that the probability of matching a key is zero, then this reduces to the expected-case complexity of solving a point location problem for  $n + 1$  disjoint subintervals of the unit interval. Consider any binary search tree whose leaves correspond to the intervals. It is easy to see that the expected number of comparisons is given by the *weighted external path length* [14] of the tree, where the weight of a leaf is the probability of the query point lying in the associated interval.

Let  $p_i$  denote the probability of falling in the  $i$ th interval. A fundamental information theoretic result due to Shannon implies that the weighted path length of any binary tree (and hence the expected number of comparisons) is at least the *entropy* of the probability distribution

$$\sum_i p_i \log\left(\frac{1}{p_i}\right).$$

(Unless otherwise stated, all logarithms are base 2.) Knuth [13] shows how to construct an optimum binary search tree in  $O(n^2)$  time using dynamic programming. Hu and Tucker [11] presented a bottom-up construction of the tree, which takes  $O(n \log n)$  time, but is quite complex. Mehlhorn [17] gives a simple construction of a binary search tree whose weighted path length is within a constant additive factor of the entropy-based lower bound. It is eminently natural to ask whether these results can be extended to planar subdivisions. To the best of our knowledge, this is the first paper to address this obvious and fundamental problem.

Consider a polygonal subdivision  $S$ . Given a region  $z$  in  $S$ , let  $p_z$  denote the probability that the query point lies inside region  $z$ . Define the *entropy* of  $S$  to be

$$\text{entropy}(S) = \sum_{z \in S} p_z \log\left(\frac{1}{p_z}\right).$$

The coordinates of the query points are assumed to be sampled independently from probability distributions over bounded intervals of the  $x$ -axes and  $y$ -axes. Both  $S$  and the probability distributions are assumed to satisfy some additional weak assumptions (see Section 2 for formal definitions).

Shannon's lower bound applies to the planar point location problem as well. We present two algorithms for the planar point location problem. The first uses quadratic space and can answer point location queries in  $2 \text{ entropy}(S) + O(1)$  expected number of point-line comparisons (i.e., given a point and a directed line, one has to determine whether the point lies to the left of, on, or to the right of the line). The second uses  $O(n)$  space, and can answer point location queries in nearly  $4 \text{ entropy}(S) + O(1)$  expected number of point-line comparisons.

The paper is organized as follows. In Section 2 we present definitions and state our results formally. In Section 3 we present background on the planar point location problem. In Section 4 we present our algorithms for the case of uniformly distributed query points, and in Section 5 we generalize our results to a wider class of probability distributions.

## 2 Definitions and Main Results

Let  $I$  and  $J$  be two arbitrary intervals of real numbers. In this paper, we only work with planar subdivisions that partition an underlying rectangle  $I \times J$  into disjoint connected regions. We allow the underlying rectangle to be the infinite plane, in which case  $I = J = (-\infty, \infty)$ .

Given a query point  $q$ , let  $x_q$  and  $y_q$  denote its  $x$  and  $y$  coordinate. Throughout this paper, we assume that  $x_q$  and  $y_q$  are two *independent* random variables. We denote the probability distribution function for  $x_q$  by  $P : I \rightarrow [0, 1]$  and the probability distribution function for  $y_q$  by  $Q : J \rightarrow [0, 1]$ . That is,  $P(x)$  is the probability that the random variable  $x_q$  is less than or equal to  $x$ , and  $Q(y)$  is the probability that the random variable  $y_q$  is less than or equal to  $y$ . We call  $(P, Q)$  a *well-behaved distribution* if  $P$  and  $Q$  are *continuous* and *strictly increasing*. For example, if  $I \times J$  is the unit square, then picking  $x_q$  and  $y_q$  uniformly and independently from  $[0, 1]$  yields a well-behaved distribution.

Let  $U$  be the unit square  $[0, 1]^2$ . We define a mapping  $f_{PQ}$  from  $I \times J$  (call this *geometric space*) to  $U$  (call this *probability space*) as follows:

$$f_{PQ}(x, y) = (P(x), Q(y)).$$

If  $(P, Q)$  is well-behaved, then  $f_{PQ}$  is a bijection as  $P$  and  $Q$  are strictly increasing. We can also generalize  $f_{PQ}$  in the obvious way for a set of points. Let  $A$  be any set points in  $I \times J$ . Then

$$f_{PQ}(A) = \{(P(x), Q(y)) : (x, y) \in A\}.$$

In this paper, we assume that each evaluation of  $P$ ,  $Q$ ,  $P^{-1}$ , and  $Q^{-1}$  takes constant time.

We are now ready to state the main results of this paper.

**Theorem 1.** *Let  $I$  and  $J$  be two intervals of real numbers. Let  $\mathcal{S}$  be a planar subdivision of  $I \times J$  of  $n$  vertices. Suppose that a well-behaved distribution  $(P, Q)$  is given for the coordinates of the query point, and for each region  $z \in \mathcal{S}$ ,  $f_{PQ}(z)$  has at most a constant number of holes and the perimeter of  $f_{PQ}(z)$  is bounded by a constant. Then*

- (i) Using  $O(n^2)$  space and  $O(n^2)$  preprocessing time, it is possible to answer point location queries in  $2 \text{entropy}(\mathcal{S}) + O(1)$  expected number of point-line comparisons.
- (ii) Using  $O(n)$  space and  $O(n \log n)$  preprocessing time, it is possible to answer point location queries in  $(4+O(1/\sqrt{\log n})) \text{entropy}(\mathcal{S}) + O(1)$  expected number of point-line comparisons.

Clearly this theorem applies if  $I \times J$  is the unit square  $U$  and  $x_q$  and  $y_q$  are chosen uniformly and independently from  $[0, 1]$ . Thus we have the following theorem, which is an interesting special case of Theorem \blacksquare

**Theorem 2.** *Let  $\mathcal{S}$  be a planar subdivision of  $U$  of  $n$  vertices. Suppose that the coordinates of the query point are chosen uniformly and independently from  $[0, 1]$ , and for each region  $z$  in  $\mathcal{S}$ ,  $z$  has at most a constant number of holes, and the perimeter of  $z$  is bounded by a constant. Then*

- (i) Using  $O(n^2)$  space and  $O(n^2)$  preprocessing time, it is possible to answer point location queries in  $2 \text{entropy}(\mathcal{S}) + O(1)$  expected number of point-line comparisons.
- (ii) Using  $O(n)$  space and  $O(n \log n)$  preprocessing time, it is possible to answer point location queries in  $(4+O(1/\sqrt{\log n})) \text{entropy}(\mathcal{S}) + O(1)$  expected number of point-line comparisons.

**Remark:** It is worth noting that any convex polygon in the geometric space is mapped by  $f_{PQ}$  to a region in the probability space that has bounded perimeter. This follows from the fact that any monotonic increasing (resp. decreasing) curve in the geometric space maps to a monotonic increasing (resp. decreasing) curve in the probability space. And the length of any monotonic curve in the unit square is bounded by 2. Thus Theorem \blacksquare applies to any subdivision of the plane into (bounded and unbounded) convex polygons.

### 3 Background

For the planar point location problem, let  $n$  denote the number of vertices in the subdivision. The early work of Dobkin and Lipton [6] showed that a query time of  $O(\log n)$  and space  $O(n^2)$  could be achieved. Lipton and Tarjan [15] showed that the space requirement could be reduced to  $O(n)$ , but their approach was rather impractical. Since then a number of more practical methods have been proposed. These include Kirkpatrick's clever hierarchical method [12], the separator method by Edelsbrunner et al. [8], the persistent search tree method by Sarnak and Tarjan [21], and the randomized incremental method by Mulmuley [20]. All of these are based on worst-case analyses. Recently Adamy and Seidel [1] presented an  $O(n)$  space data structure that achieves a worst-case query time of  $\log n + 2\sqrt{\log n} + O(\log^{1/4} n)$  point-line comparisons, thus approaching the worst-case information theoretic lower bound.

Existing work on expected case-performance has been based on the assumption that both the subdivision and the queries satisfy certain probabilistic assumptions. Edahiro et al. [7] proposed a practical algorithm for planar point location based on bucketing techniques. Their method may use  $\Theta(n^2)$  space in the worst case. Methods using kd-trees, quad-trees, and R-trees are also popular in practice, but their analyses do not hold in the worst case. Mucke et al. [19] and Devroye et al. [5] have analyzed methods based on walking through subdivisions. For Delaunay triangulations of uniformly distributed data sets, these methods take expected time close to  $O(n^{1/3})$  and  $O(n^{1/4})$  in two and three dimensions, respectively.

Goodrich et al. [10] presented an interesting point location method, which adapts to the query distribution. Intuitively, if a cell is accessed more frequently, then the data structure is modified to ensure that the time for subsequent accesses to the cell is reduced. They show that the amortized time complexity for accessing cell  $i$  in a sequence of  $m$  queries is  $O(\min\{\log n, \log(t(i) + 1), \log(m/f(i))\})$ , where  $t(i)$  is the number of different queries between two accesses to cell  $i$ , and  $f(i)$  is the frequency of accesses to cell  $i$ . A limitation of their approach is that the cells are not the regions in the given subdivision; instead they are the trapezoids in the refined subdivision formed by passing a vertical line through each segment endpoint. This can adversely affect the query time.

## 4 The Uniform Distribution Case

We first present our techniques in attacking the case when  $I \times J$  is  $U$  and the coordinates of the query point are chosen uniformly and independently from  $[0, 1]$ . The techniques are based on certain box decompositions of the planar subdivision  $\mathcal{S}$  of  $U$ . In the case of general well-behaved distributions  $(P, Q)$ , the key insight is that the map  $f_{PQ}$  transforms the problem in the geometric space to a problem in the probability space, where we are to locate query points in the subdivision  $f_{PQ}(\mathcal{S})$  of  $U$ , and the coordinates of the query point are chosen uniformly and independently from  $[0, 1]$ . Thus, for general well-behaved distributions, it suffices to invoke the techniques for uniform distribution in the probability space to organize a point location data structure. Given a query point  $q$ , we use this data structure to locate the region  $z'$  in  $f_{PQ}(\mathcal{S})$  containing  $f_{PQ}(q)$ , and the region in  $\mathcal{S}$  containing  $z$  is then given by  $f_{PQ}^{-1}(z')$ . These claims will be proved formally in Section 5.

In the following, we focus on the uniform distribution case. We first present an algorithm, which uses  $2 \text{entropy}(\mathcal{S}) + O(1)$  expected number of point-line comparisons. The data structure needs  $O(n^2)$  space and can be built in  $O(n^2)$  time. Later we present another algorithm which reduces the space to  $O(n)$  and the preprocessing time to  $O(n \log n)$ . The expected number of point-line comparisons goes up to nearly  $4 \text{entropy}(\mathcal{S}) + O(1)$ .

A lemma proved in [2] will be very useful. We state it in a form which is applicable in two dimensions. The result concerns with overlaying two planar subdivisions of  $U$ . One subdivision is the given planar subdivision  $\mathcal{S}$  of  $U$ . The

other subdivision is a decomposition of  $U$  into *cells* that enjoys the following properties, for some constants  $c_a$  and  $c_n$ :

- (A.1) *Difference of Two Rectangles*: A cell is the set-theoretic difference of two axis-parallel rectangles, one enclosed within the other. We call these the *outer rectangle* and *inner rectangle* of the cell. Note that the inner rectangle need not be present. Given a cell  $u$ , we let  $u_O$  and  $u_I$  denote its outer and inner rectangle, respectively. Also, we define the size of  $u$ , denoted by  $s_u$ , to be the length of the longest side of  $u_O$ .
- (A.2) *Bounded Aspect Ratio*: The outer rectangle and inner rectangle (if present) have aspect ratio (ratio of longest to shortest side) bounded by  $c_a$ . (In this case we say that the cell has aspect ratio at most  $c_a$ .)
- (A.3) *Stickiness*: If the cell has an inner rectangle, then for each dimension, the separation between the corresponding faces of the inner and outer rectangle is either 0 or at least the length of the inner rectangle along that dimension.
- (A.4) *Proximity to  $S$* : For each cell  $u$ , there is some edge or vertex in  $S$  within a distance of  $c_n \cdot s_u$  from any point in  $u_O$ .
- (A.5) *Disjointness*: Given any two cells, either the outer rectangles of the two cells are disjoint or the outer rectangle of one cell is contained within the inner rectangle of the other.

We define a *fragment* to be a connected component in the intersection between a cell in the decomposition and a region in  $S$ . Let  $\mathcal{F}$  be the set of all fragments. Let  $\text{area}(x)$  denote the area of region  $x$ .

**Lemma 1.** *Let  $S$  be a planar subdivision of  $U$  such that each region has at most a constant number of holes and the total boundary length of each region is bounded by a constant  $c_s$ . Let  $\mathcal{D}$  be a decomposition of  $U$  that satisfies properties A.1, A.2, A.3, A.4, and A.5. Let  $\mathcal{F}$  be the set of fragments in the overlay of  $S$  and  $\mathcal{D}$ . Then*

$$\sum_{x \in \mathcal{F}} \text{area}(x) \log \frac{1}{\text{area}(x)} \leq 2 \sum_{z \in S} \text{area}(z) \log \frac{1}{\text{area}(z)} + O(1),$$

where the constant in the  $O$ -notation depends on  $c_a$ ,  $c_s$ , and  $c_n$ .

#### 4.1 Quadratic Space Solution

We prove Theorem 2(i) in this subsection. Let  $S$  be the given planar subdivision of  $U$  such that each region has at most a constant number of holes, and the total boundary length of each region is bounded by a constant. We construct a hierarchical decomposition of  $U$  by building a box-decomposition tree (BD-tree) on the vertices of  $S$  [4, 22]. Initially, the BD-tree contains only one node which is the root. Each node represents a cell and the root represents  $U$ . We keep expanding the tree until some terminating condition is satisfied. The leaf cells form the desired decomposition of  $U$ . We describe how to construct children for a node  $u$  below. For convenience, we also use  $u$  to denote the cell it represents.

If  $u$  contains at most one vertex, then  $u$  is a leaf cell. Otherwise, it can be guaranteed inductively that  $u$  is a rectangle and we recursively construct two children of  $u$  as follows. Split  $u$  orthogonally at the midpoint of its longest side to obtain two rectangles  $v$  and  $w$ . If both  $v$  and  $w$  contain some vertex, then we make  $v$  and  $w$  children of  $u$ . This operation is called a *midpoint split*. Otherwise, if  $v$  or  $w$  is empty, then we recursively apply the midpoint splitting rule to the non-empty rectangle, until we obtain a rectangle  $v'$  such that  $v'$  will be split into two non-empty rectangles. We make  $v'$  and  $u \setminus v'$  children of  $u$ . This operation is called a *shrink*. Note that  $u \setminus v'$  is a leaf cell and it contains no vertex.

To construct the tree efficiently, we use a standard trick due to Vaidya [22] for partitioning the points. We store the data points contained in a cell in  $d$  separate lists, each sorted by one of the coordinates, that are cross-referenced with each other. Instead of updating the lists after each split, we update them after a sequence of splits is performed, until each of the resulting subsets contains fewer than half the initial number of points. Also, assuming a model of computation in which exclusive-or, integer floor, powers of 2, and integer logarithm can be computed on point coordinates, the shrink operation can be performed in  $O(d)$  time. (For example, see Bern [3]). Straightforward modification of the argument given by Vaidya leads to a construction time of  $O(n \log n)$ . (We mention that we can achieve the same construction time without using non-algebraic operations by building the *sliding-midpoint tree* [16, 18] instead. It can be shown that Lemma II holds for the fragments induced by the leaves of the sliding-midpoint tree. The query algorithm and the rest of the analysis given in this section can also be easily adapted.)

The cells associated with the leaves of the BD-tree satisfy properties A.1, A.2 ( $c_a = 2$ ), A.3, A.4 ( $c_n = 2$ ), and A.5. In addition, the BD-tree has the following property, which is important for our analysis.

**Lemma 2.** *Let  $\mathcal{T}$  be a BD-tree constructed on some point set in  $U$ . For any query point  $q$ , the number of point-line comparisons needed in traversing  $\mathcal{T}$  to locate the leaf cell  $y$  containing  $q$  is at most  $\log \frac{1}{\text{area}(y)} + O(1)$ .*

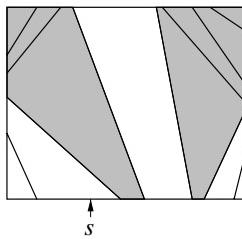
*Proof.* Suppose that we arrive at a node representing a cell  $u$  in traversing  $\mathcal{T}$  and we need to decide which of its child cells should be visited. Let  $v$  and  $w$  be its child cells. If  $v$  and  $w$  are formed by a midpoint split, then one point-line comparison is needed to determine whether  $v$  or  $w$  contains  $q$ . Note that the area of  $v$  and  $w$  are both half the area of  $u$ . If  $v$  and  $w$  are formed by a shrink, then one is a leaf cell, say  $v$ , and it encloses the other child cell, say  $w$ . Let  $i$ ,  $1 \leq i \leq 4$ , denote the number of sides that the inner box of  $v$  does not share with the boundary of  $u$ . Thus, it takes  $i$  point-line comparisons to decide whether  $v$  or  $w$  contains the query point  $q$ . Note that the area of  $w$  is at most  $1/2^i$  times the area of  $u$ , for  $2 \leq i \leq 4$ . Therefore, in both cases of midpoint split or shrink, if we spend  $i$  point-line comparisons to decide the next child cell to visit and this child cell is not a leaf cell, then the area of this child cell is at most  $1/2^i$  times the area of its parent. The area of  $U$  is 1. Therefore, the number of point-line comparisons needed to reach the leaf cell  $y$  containing  $q$  is

at most  $\log(1/\text{area}(y)) + O(1)$ , where the  $O(1)$  additive term comes from the last  $i$  point-line comparisons,  $1 \leq i \leq 4$ , spent at the parent of  $y$ .

Let  $y$  denote any leaf cell of the BD-tree. Observe that  $y$  is either a rectangle containing at most one vertex of  $\mathcal{S}$ , or it is the set-theoretic difference of an outer and inner rectangle, in which case it contains no vertex of  $\mathcal{S}$ . In each case we partition  $y$  into at most four rectangles whose interior contains no vertex of  $\mathcal{S}$  (we call them *subcells*). If  $y$  is a rectangle and contains no vertex of  $\mathcal{S}$  in its interior, then the subcell is  $y$  itself. Otherwise if it contains a vertex of  $\mathcal{S}$  in its interior, then we split it into two subcells by a vertical line passing through this vertex. Otherwise it must be the set-theoretic difference of an outer and inner rectangle. In this case we partition it into at most four subcells by passing lines coinciding with the vertical sides of the inner rectangle.

Define a *pseudo-fragment* to be a connected component in the intersection of any subcell with a region in  $\mathcal{S}$ . Clearly each fragment is partitioned into at most four pseudo-fragments. Let  $z$  be any subcell. Observe that  $z$  contains no vertex of  $\mathcal{S}$  and intersects  $O(n)$  edges of the subdivision  $\mathcal{S}$ . Thus  $z$  is partitioned into at most  $O(n)$  pseudo-fragments. Since the subdivision inside  $z$  is so simple, we can locate the pseudo-fragment in  $z$  containing the query point by searching an auxiliary structure associated with  $z$ .

If there is an edge that intersects two opposite sides of  $z$ , then let  $s$  be one of the sides intersected. The edges intersecting  $s$  divide  $z$  into super-fragments which can be linearly ordered along  $s$ . (See Figure 11) Each super-fragment is either a pseudo-fragment by itself, or it is further subdivided by other edges into pseudo-fragments which can be linearly ordered within the super-fragment. (There are at most two super-fragments which are further subdivided; these are shown shaded in the figure.) Thus, we first organize a weighted search tree [17] for the super-fragments with their area as weights. Each super-fragment points to another weighted search tree storing the linearly ordered pseudo-fragments within the super-fragment (the area of the pseudo-fragments are the weights in this second level tree). If there is no edge that intersects two opposite sides of  $z$ , we can do the above using any side  $s$  of  $z$ .



**Fig. 1.** Super-fragments inside a subcell.

A single query is now answered by first locating the leaf cell in the BD-tree that contains the query point  $q$ . Then we determine which of the at most

four subcells associated with this leaf cell contains  $q$ . Then we query the auxiliary structure for the subcell to locate the pseudo-fragment containing  $q$ . Each pseudo-fragment lies inside a region in  $\mathcal{S}$  and hence we have the solution to the query.

We analyze the time to answer a single query as follows. By Lemma 2, the number of point-line comparisons needed to reach a leaf cell  $y$  of the BD-tree is  $\log(1/\text{area}(y)) + O(1)$ . It takes  $O(1)$  point-line comparisons to find the subcell  $z$  containing  $q$ . Then we query the auxiliary structure associated with  $z$ . It is known [17] that querying a weighted search tree takes at most  $\log(K/k) + 2$  comparisons, where  $K$  is the total weight of all the items, and  $k$  is the weight of the item being searched for. Therefore, querying the auxiliary structure takes  $\log(\text{area}(z)/\text{area}(z')) + \log(\text{area}(z')/\text{area}(x)) + 4$  point-line comparisons, where  $z'$  and  $x$  are the super-fragment and pseudo-fragment containing the query point, respectively. Hence, the total number of point-line comparisons is at most  $\log(1/\text{area}(y)) + \log(\text{area}(z)/\text{area}(z')) + \log(\text{area}(z')/\text{area}(x)) + O(1) \leq \log(1/\text{area}(x)) + O(1)$ .

The probability of the query point lying in a pseudo-fragment  $x$  is clearly  $\text{area}(x)$ . Thus, the expected number of point-line comparisons to answer a query is at most

$$\sum_{x \in \mathcal{F}'} \text{area}(x) \left( \log \frac{1}{\text{area}(x)} + O(1) \right) = \text{entropy}(\mathcal{F}') + O(1),$$

where  $\mathcal{F}'$  is the set of pseudo-fragments. Since each fragment is partitioned into at most four pseudo-fragments, it is easy to see that  $\text{entropy}(\mathcal{F}') = \text{entropy}(\mathcal{F}) + O(1)$ . Therefore, the expected number of point-line comparisons is at most  $\text{entropy}(\mathcal{F}) + O(1)$ , which is at most  $2 \text{entropy}(\mathcal{S}) + O(1)$  by Lemma 11.

We analyze the space of the entire data structure. The space needed by the BD-tree is  $O(n)$ . Since there are  $O(1)$  subcells for each leaf cell, and  $O(n)$  pseudo-fragments for each subcell, the auxiliary structure at each leaf cell also takes  $O(n)$  space. Thus, the total space is  $O(n^2)$ . As mentioned earlier the BD-tree can be constructed in  $O(n \log n)$  time. A weighted search tree of  $m$  sorted items can be constructed in  $O(m)$  time [17]. Thus, the auxiliary structure at each leaf cell can be constructed in  $O(n)$  time which leads to a total preprocessing time of  $O(n^2)$ . This completes the proof of Theorem 2(i).

## 4.2 Linear Space Solution

We prove Theorem 2(ii) in this subsection. First, we also build a decomposition tree on the vertices of  $\mathcal{S}$ , but it is different from the BD-tree in the quadratic space solution. The cell at each node of the tree will be rectangles of bounded aspect ratio. We will classify the leaf cells of the tree into two types, *S-type* and *L-type*. The root of the tree represents the unit square  $U$ . Inductively suppose that we are to construct the children of a cell  $u$ .

1. If  $\text{area}(u) < 1/n$ , we label  $u$  an  $S$ -type leaf cell.
2. If  $\text{area}(u) \geq 1/n$  and  $u$  intersects no edge of  $\mathcal{S}$ , then  $u$  must be completely contained in some region of  $\mathcal{S}$ ; we store the name of this region with  $u$ . In addition, we label  $u$  an  $L$ -type leaf.
3. If  $\text{area}(u) \geq 1/n$ ,  $u$  intersects some edge of  $\mathcal{S}$ , and each region in  $u \cap \mathcal{S}$  has area less than  $1/n$ , then we label  $u$  an  $S$ -type leaf cell
4. Otherwise,  $\text{area}(u) \geq 1/n$ ,  $u$  intersects some edge of  $\mathcal{S}$ , and some region in  $u \cap \mathcal{S}$  has area at least  $1/n$ . We split  $u$  using a midpoint split into two cells  $v$  and  $w$ , and make them children of  $u$ . Then we recursively construct the descendants of  $v$  and  $w$ .

We denote this decomposition tree by  $\mathcal{T}(\mathcal{S})$ . The cells associated with the leaves of the tree satisfy properties A.1, A.2 ( $c_a = 2$ ), A.3, A.4 ( $c_n = 2$ ), and A.5. We also have the following result which is analogous to Lemma 2.

**Lemma 3.** *For any query point  $q$ , the number of point-line comparisons needed in traversing  $\mathcal{T}(\mathcal{S})$  to locate the leaf cell  $y$  containing  $q$  is at most  $\log \frac{1}{\text{area}(y)} + O(1)$ .*

The final step of preprocessing is to construct the worst-case planar point location data structure for  $\mathcal{S}$  invented by Adamy and Seidel [1]. This data structure uses  $O(n)$  space and can be constructed in  $O(n \log n)$  time. A point location query can be answered using  $\log n + 2\sqrt{\log n} + O(\log^{1/4} n)$  point-line comparisons.

Given a query point  $q$ , we first descend  $\mathcal{T}(\mathcal{S})$  to find the leaf cell  $x$  containing  $q$ . If  $x$  is an  $L$ -type leaf cell, then we report the region of  $\mathcal{S}$  containing  $x$  and terminate. Otherwise,  $x$  is an  $S$ -type leaf cell, and we simply resort to Adamy and Seidel's data structure to answer the point location query.

**Space analysis.** Each leaf cell of  $\mathcal{T}(\mathcal{S})$  has area at least  $1/2n$  and they partition the unit square  $U$ . This implies that  $\mathcal{T}(\mathcal{S})$  has  $O(n)$  leaves and hence  $O(n)$  nodes. Adamy and Seidel's data structure use  $O(n)$  space. Thus, the total space needed is  $O(n)$ .

**Query time analysis.** Recall that a *fragment* is a connected component of the intersection of the leaf cells of  $\mathcal{T}(\mathcal{S})$  and  $\mathcal{S}$ . Let  $\mathcal{F}$  denote the set of all fragments. By Lemma 1, we have  $\text{entropy}(\mathcal{F}) \leq 2 \text{entropy}(\mathcal{S}) + O(1)$ . In the following, we show that the expected number of point-line comparisons to answer a query is  $(2 + O(1/\sqrt{\log n})) \text{entropy}(\mathcal{F})$  and so the desired bound of  $(4 + O(1/\sqrt{\log n})) \text{entropy}(\mathcal{S}) + O(1)$  follows.

We call a fragment *large* if its area is at least  $1/n$ , and *small* otherwise. Let  $\mathcal{F}_1$  and  $\mathcal{F}_2$  denote the set of large and small fragments, respectively. A large fragments is exactly an  $L$ -type leaf cell and vice versa. Small fragments lie inside  $S$ -type leaves.

We analyze the time to locate a query point  $q$ . Suppose that  $q$  lies inside a fragment  $x \in \mathcal{F}$ . If  $x$  is large, then  $x$  is an  $L$ -type leaf, and the number of comparisons needed to reach  $x$  is  $\log(1/\text{area}(x))$  by Lemma 3. If  $x$  is

small, then the query procedure will first locate the leaf cell  $y$  containing  $x$  and then query the worst-case data structure. By Lemma 3, the number of point-line comparisons needed to reach  $y$  is  $\log(1/\text{area}(y))$ . Since  $\text{area}(y) \geq \text{area}(x)$ ,  $\log(1/\text{area}(y)) \leq \log(1/\text{area}(x))$ . Adding the number of point-line comparisons needed for querying the worst-case data structure, the total number of comparisons is at most  $\log(1/\text{area}(x)) + \log n + 2\sqrt{\log n} + O(\log^{1/4} n)$ . Since  $x$  is small,  $\text{area}(x) < 1/n$  which implies that  $\log(1/\text{area}(x)) > \log n$ . So the total number of comparisons is at most  $(2 + O(1/\sqrt{\log n})) \log(1/\text{area}(x))$ .

The probability that  $q$  lies in a fragment  $x$  is clearly  $\text{area}(x)$ . Thus, the expected number of point-line comparisons to answer a query is bounded by

$$\begin{aligned} & \sum_{x \in \mathcal{F}_1} \text{area}(x) \log \frac{1}{\text{area}(x)} + \sum_{x \in \mathcal{F}_2} \left( 2 + O\left(\frac{1}{\sqrt{\log n}}\right) \right) \text{area}(x) \log \frac{1}{\text{area}(x)} \\ & \leq \left( 2 + O\left(\frac{1}{\sqrt{\log n}}\right) \right) \text{entropy}(\mathcal{F}). \end{aligned}$$

**Preprocessing time.** When we construct the child cells of a cell  $u$  during preprocessing, the most time consuming part of the construction is to determine whether each region in  $u \cap \mathcal{S}$  has area less than  $1/n$ . We describe a method to carry out this computation efficiently.

Define  $L_u$  to be the set of regions of area at least  $1/n$  in  $u \cap \mathcal{S}$ . The observation is that any region in  $L_v$  at a child  $v$  of  $u$  must be contained in some region in  $L_u$ . Therefore, our strategy is to compute  $L_u$  for each node  $u$  inductively.

Let  $r$  be the root of  $\mathcal{T}(\mathcal{S})$  and so  $r \cap \mathcal{S} = \mathcal{S}$ . We simply traverse  $\mathcal{S}$  in  $O(n)$  time to collect all regions of area at least  $1/n$  in  $L_r$ . Inductively, let  $v$  be the child of  $u$  and we are to compute  $L_v$ . For each region  $z$  in  $L_u$ , we claim that we can compute the intersection  $z \cap v$  in time proportional to the size of  $z$ . (Note that  $z \cap v$  may consist of several connected components.)

This can be done by clipping  $z$  with four halfplanes successively. We describe the first clipping as follows. Let  $\ell$  be the bounding line of a halfplane. For convenience, denote the size of  $z$  by  $|z|$ . First, compute the  $O(|z|)$  intersections between  $\ell$  and the boundary of  $z$  in  $O(|z|)$  time by brute-force. Second, apply Jordan sorting to sort these intersections in order of their appearance on  $\ell$ . This can also be done in  $O(|z|)$  time [9]. Third, start a clockwise traversal from some vertex of  $z$  within the halfplane. If we come to an intersection on  $\ell$ , then we use the sorted list of intersections to jump to the next intersection along  $\ell$ . The traversal stops when we come back to a visited vertex, and we have traversed the boundary of one connected component of the clipped  $z$ . Then we repeat the traversal from an unvisited vertex of  $z$  within the halfplane and so on until no such vertex is left. This traverses all connected components in the clipped  $z$ . Since each vertex of  $z$  and each intersection on  $\ell$  is visited at most once, this also takes  $O(|z|)$  time. This completes the first clipping. Each subsequent clipping is done the same way. Since we have added at most  $O(|z|)$  new vertices after a clipping and there are four clippings, we conclude that each clipping takes  $O(|z|)$  time.

After obtaining  $z \cap v$ , we can then retain only the components in  $z \cap v$  that has area at least  $1/n$  and include them in  $L_v$ . Repeating this for each region in  $L_u$  yields  $L_v$ . The total time needed is then proportional to the sum of sizes of regions in  $L_u$ . Let  $E_i$  denote the set of edges on the boundaries of regions in  $L_u$  for all nodes  $u$  at level  $i$  of  $\mathcal{T}(\mathcal{S})$ . We claim that for any level  $i$ , the number of edges in  $E_i$  is  $O(n)$ . Since  $\mathcal{T}(\mathcal{S})$  is constructed using midpoint split and each leaf cell has area at least  $1/2n$ , the number of levels in the tree is  $O(\log n)$ , and it follows that the preprocessing time is  $O(n \log n)$ .

To see that there are  $O(n)$  edges in  $E_i$ , note that there are two categories of edges in  $E_i$ . The first category consists of edges that lie on the sides of a cell, and the second category consists of edges that lie on edges of  $\mathcal{S}$ . Observe that edges in the first category can be charged against edges in the second category, so we only need to show that the number of edges in the second category is  $O(n)$ . The second category can be divided into two groups. The first group consists of edges that are incident to a vertex of  $\mathcal{S}$  inside a cell at level  $i$ , and the second group consists of the remaining edges. It is clear that the number of edges of the first group can be no more than the total degree of the vertices of  $\mathcal{S}$ , which is  $O(n)$ . To count the number of edges of the second group, first observe that the number of regions with area at least  $1/n$  in cells at level  $i$  is at most  $n$ . Second, each such region can have at most four boundary edges that are not incident to a vertex of  $\mathcal{S}$  inside a cell at level  $i$ . Thus the number of edges of the second group is at most  $4n$ . Hence the total number of edges in  $E_i$  is  $O(n)$ , which is the desired claim.

## 5 General Well-Behaved Distributions

Given a planar subdivision  $\mathcal{S}$  and a well-behaved distribution  $(P, Q)$ , our main idea is that we can organize our point location data structure (the quadratic space version or linear space version) in Theorem 2 in the probability space. Then given a query point  $q$ , we locate the region  $z'$  in  $f_{PQ}(\mathcal{S})$  containing  $f_{PQ}(q)$  and then return  $f_{PQ}^{-1}(z')$ .

For the above strategy to work, there are several requirements. First, the  $x$  and  $y$  coordinates of  $f_{PQ}(q)$  should be uniformly and independently chosen from  $[0, 1]$ . Second,  $f_{PQ}(\mathcal{S})$  is a planar subdivision, and each region has at most a constant of holes and the perimeter of each region is bounded by a constant. Third, if we were to apply Theorem 2 directly, we would require that  $f_{PQ}(\mathcal{S})$  be a polygonal planar subdivision, but this is usually untrue. Instead, we will map back and forth between the geometric and probability spaces using  $f_{PQ}$  and  $f_{PQ}^{-1}$  to construct and query our data structure in the probability space. We describe below how these requirements are satisfied.

Let  $x'_q$  be the  $x$ -coordinate of  $f_{PQ}(q)$ . The probability  $\text{prob}(x'_q \leq x')$  that  $x'_q$  is less than or equal to  $x'$  for some  $0 \leq x' \leq 1$  is equal to  $\text{prob}(P(x_q) \leq x')$ , where  $x_q$  is the  $x$ -coordinate of  $q$ . But  $\text{prob}(P(x_q) \leq x') = \text{prob}(x_q \leq P^{-1}(x'))$  which is equal to  $P(P^{-1}(x')) = x'$  by definition of  $P$ . So  $\text{prob}(x'_q \leq x') = x'$  and

$x'_q$  is uniformly picked from  $[0, 1]$ . Similarly, we can show that the  $y$ -coordinate of  $f_{PQ}(q)$  is uniformly picked in  $[0, 1]$ .

Given two real numbers  $\alpha, \beta \in I$ , since  $P$  is strictly increasing,  $\alpha \leq \beta$  iff  $P(\alpha) \leq P(\beta)$  and equality holds exactly when  $\alpha = \beta$ . Therefore, the left-right ordering of points by  $x$ -coordinate in  $I \times J$  is preserved in  $U$  after the mapping. A similar reasoning about  $Q$  shows that the above-below ordering of points by  $y$ -coordinate is also preserved. Also, a point  $p$  is on a line segment  $\xi$  in  $I \times J$  iff  $f_{PQ}(p)$  is on  $f_{PQ}(\xi)$  in  $U$ . Thus, incidence relations in  $\mathcal{S}$  are preserved in  $f_{PQ}(\mathcal{S})$  and hence  $f_{PQ}(\mathcal{S})$  is a planar subdivision of  $U$ . In Theorem 1, it is already assumed that each region in  $f_{PQ}(\mathcal{S})$  has at most a constant number of holes, and the perimeter of each region is bounded by a constant.

We now deal with issue that  $f_{PQ}(\mathcal{S})$  may not be a polygonal planar subdivision. In constructing our data structure in  $U$ , we need to perform two primitives. The first primitive is to determine whether a vertex lies above, below, to the left, or to the right of an orthogonal line. (This is needed in shrinking.) The second primitive is to compute the intersection between an (possibly curvy) edge  $\xi'$  and an orthogonal line segment. (This is needed in midpoint split and shrinking.) Since ordering and incidence relations are preserved, these two primitives can be provided by first going back to the geometric space, perform the computation, and map the result back to the probability space. Note that a vertical/horizontal line segment is always mapped to a vertical/horizontal line segment and vice versa. Also, for the second primitive, we would be intersecting  $f_{PQ}^{-1}(\xi')$ , which must be a line segment, with an orthogonal line segment in the geometric space. This can be done in constant time in the geometric space, and then we map the result using  $f_{PQ}$  to the intersection desired in the probability space.

To see the correctness of our approach to answer a query, first observe that, by continuity of  $P$  and  $Q$ , a closed curve in  $I \times J$  is mapped by  $f_{PQ}$  to a closed curve in  $U$ . Since ordering and incidence relations are preserved, a point  $p$  lies inside/on/outside a closed curve  $\xi$  in  $I \times J$  iff  $f_{PQ}(p)$  lies inside/on/outside  $f_{PQ}(\xi)$ . Thus, given a query point  $q$  in the geometric space and a region  $z'$  in  $f_{PQ}(\mathcal{S})$  containing  $f_{PQ}(q)$ ,  $f_{PQ}^{-1}(z')$  is the region in  $\mathcal{S}$  containing  $q$ . In searching our data structure, we need to tell whether the query point  $f_{PQ}(q)$  in  $U$  lies above, below, to the left, or to the right of an orthogonal line or a curvy edge  $\xi'$ . We have seen that this can be done for an orthogonal line. For a curvy edge  $\xi'$ , we simply return the relation between  $q$  and  $f_{PQ}^{-1}(\xi')$ , which must be a line segment, in the geometric space. This establishes the correctness of our approach to answer a query.

In all, Theorem 1 holds assuming that each evaluation of the functions  $P$ ,  $Q$ ,  $P^{-1}$ , and  $Q^{-1}$  takes constant time.

## References

- [1] U. Adamy and R. Seidel. Planar point location close to the information-theoretic lower bound. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, 1998.
- [2] S. Arya and H. Y. Fu. Expected-case complexity of approximate nearest neighbor searching. In *Proc. 11th ACM-SIAM Sympos. Discrete Algorithms*, pages 379–388,

2000. Extended version appears as HKUST Technical Report HKUST-TCSC-2000-03, URL: <http://www.cs.ust.hk/tcsc/RR>.
- [3] M. Bern, D. Eppstein, and S.-H. Teng. Parallel construction of quadtrees and quality triangulations. In *Proc. 3rd Workshop Algorithms Data Struct.*, volume 709 of *Lecture Notes in Computer Science*, pages 188–199. Springer-Verlag, 1993.
  - [4] P. B. Callahan and S. R. Kosaraju. A decomposition of multi-dimensional point-sets with applications to  $k$ -nearest-neighbors and  $n$ -body potential fields. In *Proc. 24th Ann. ACM Sympos. Theory Comput.*, pages 546–556, 1992.
  - [5] L. Devroye, E. P. Mücke, and B. Zhu. A note on point location in Delaunay triangulations of random points. *Algorithmica*, 22:477–482, 1998.
  - [6] D. P. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM J. Comput.*, 5:181–186, 1976.
  - [7] M. Edahiro, I. Kokubo, and T. Asano. A new point-location algorithm and its practical efficiency — Comparison with existing algorithms. *ACM Trans. Graph.*, 3(2):86–109, 1984.
  - [8] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.
  - [9] K.Y. Fung, T.M. Nicholl, R.E. Tarjan, and C.J. Van Wyk. Simplified linear-time jordan sorting and polygon clipping. *Inform. Process. Lett.*, 35:85–92, 1990.
  - [10] M. T. Goodrich, M. Orlitsky, and K. Ramaiyer. Methods for achieving fast query times in point location data structures. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 757–766, 1997.
  - [11] T. C. Hu and A. Tucker. Optimum computer search trees. *SIAM J. of Applied Math.*, 21:514–532, 1971.
  - [12] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.
  - [13] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.
  - [14] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.
  - [15] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9:615–627, 1980.
  - [16] S. Maneewongvatana and D. M. Mount. Analysis of approximate nearest neighbor searching with clustered point sets. In *ALENEX*, 1999.
  - [17] K. Mehlhorn. Nearly optimal binary search trees. *Acta Informatica*, 5:287–295, 1975.
  - [18] D. M. Mount and S. Arya. ANN: A library for approximate nearest neighbor searching. 2nd Annual CGC Workshop on Computational Geometry, URL: <http://www.cs.umd.edu/~mount/ANN>, 1997.
  - [19] E. P. Mücke, I. Sais, and B. Zhu. Fast randomized point location without pre-processing in two- and three-dimensional Delaunay triangulations. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 274–283, 1996.
  - [20] K. Mulmuley. A fast planar partition algorithm, I. *J. Symbolic Comput.*, 10(3–4):253–280, 1990.
  - [21] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986.
  - [22] P. M. Vaidya. An  $O(n \log n)$  algorithm for the all-nearest-neighbors problem. *Discrete Comput. Geom.*, 4:101–115, 1989.

# A New Competitive Strategy for Reaching the Kernel of an Unknown Polygon

Leonidas Palios

University of Ioannina, GR-45110 Ioannina, Greece  
[palios@cs.uoi.gr](mailto:palios@cs.uoi.gr)

**Abstract.** We consider the following motion planning problem for a point robot inside a simple polygon  $P$ : starting from an arbitrary point  $s$  of  $P$ , the robot aims at reaching the closest point  $t$  of  $P$  from where the entire polygon  $P$  can be seen; the robot does not have complete knowledge of  $P$  but is equipped with a 360-degree vision system that helps it “see” its surrounding space. We are interested in a competitive path planning algorithm, i.e., one that produces a path whose length does not exceed a constant  $c$  times the length of the shortest off-line path (in this case,  $c \times \text{distance}(s, t)$ ); the constant  $c$  is called the competitive factor. In this paper, we present a new strategy that achieves a competitive factor of  $\sim 3.126$ , improving over a 4.14-competitive strategy of Icking and Klein and a 3.829-competitive strategy of Lee et al. Our strategy possesses two additional advantages: first, the first point reached from where the entire polygon  $P$  is seen is precisely the closest such point to the starting position  $s$ , and second, all the points of the path are directly determined in terms of  $s$  and of polygon vertices, which implies that an actual robot following the strategy is not expected to deviate much from its course due to numerical error. The competitiveness analysis is based on properties of the class of curves with increasing chords.

**Keywords:** Motion planning, competitive algorithm, kernel, simple polygon, curve with increasing chords.

## 1 Introduction

The field of robot motion planning has received considerable attention during the 1980s, but research intensified in the late 1980s when technological advances allowed the autonomous function of robots. This, along with the need for autonomous robots to undertake tasks that may be dangerous for humans (areas polluted by chemicals, space exploration, etc.), led to a number of results pertaining to motion planning problems in unknown or partially known environments (see [4] for a survey). The general motion planning problem for an autonomous robot involves devising a strategy which can help the robot to get to a destination point in an environment which is being “discovered” by means of a vision system (or tactile sensing in some early work). Most motion planning problems are being modeled as two-dimensional problems where the robot is a point moving inside or around polygonal shapes. This is not really restrictive, as real-world

problems can be reduced to this formulation by means of transformations of the geometric boundaries of the objects in the robot's world (Minkowski sum, etc).

Of course, one is interested in having strategies which guarantee that the path traveled by the robot up to its destination is no more than a constant times the length of the shortest path if the environment was completely known. Such strategies are called *competitive* [14], and the ratio of the length of the actual path traveled over the length of the shortest path is called the *competitive factor*. In other words, the competitive strategies guarantee that the effort expended is not far from the optimal. Research results have indicated that finding competitive strategies for different motion planning problems exhibits varying degrees of difficulty (from obtaining constant competitive solutions to proving that finding a competitive solution is P-SPACE complete; see [1], [12]).

In this paper, we consider the problem of planning the path of a robot inside a polygon from any given starting position to a point from where the entire polygon can be seen; in fact, the closest such point to the starting position is sought. The robot is equipped with a 360-degree vision system. This is the problem of reaching the *kernel* of a polygon, and is what a mechanical guard is called to solve in order to position itself so that it watches its territory. The problem has been considered by Icking and Klein [5] who described a strategy to reach the closest point of the kernel at a competitive factor of  $\sim 5.48$ ; a tighter analysis by Lee and Chwa [8] showed that the strategy is  $\sim 4.14$ -competitive. Icking and Klein also showed that no competitive factor less than  $\sqrt{2}$  can be achieved. A different strategy with a competitive factor of  $\sim 3.829$  was later described by Lee et al. [9], while López-Ortiz and Schuierer [10] improved the lower bound to  $\sim 1.48$ . López-Ortiz and Schuierer also noted that the competitive factor of [5] is not guaranteed for negative instances (i.e., when the polygon has empty kernel) and described a strategy that is guaranteed to work even in this case at a competitive factor of  $\sim 46.35$ .

Our work contributes a new strategy for reaching the kernel of an unknown polygon  $P$  with nonempty kernel which achieves a competitive factor of  $\sim 3.126$ . The path consists of line segments and circular arcs whose total number is linear in the size of  $P$ . Our strategy is designed so that the robot walks into the kernel at precisely the point that is closest to the starting position; additionally, it has the advantage that any point of the course is determined by the starting position of the robot and vertices of  $P$ , and therefore an actual robot following the strategy is not expected to deviate much from its course due to accumulated numerical errors. The competitiveness analysis is based on properties of the class of curves with increasing chords [13]. Experimental results suggest that the strategy performs better than the theoretical competitive factor. (A similar strategy has been used in [7] for motion planning in a street-polygon.)

The paper is structured as follows. In Section 2 we review the terminology that we use throughout the paper, and in Section 3 we outline our strategy and state some of the properties of the resulting path. In Section 4 we establish the competitive factor of the strategy, and in Section 5 we conclude with final remarks and open questions.

## 2 Terminology

A *simple polygon* is the region enclosed by a single closed non-self-intersecting polygonal line; thus, a simple polygon does not have “holes” in it. The set of all points  $p$  of a simple polygon  $P$  such that the line segment that connects  $p$  with any other point of  $P$  lies entirely in  $P$  is called the *kernel* of the polygon. If we define the *inner halfplane* of an edge as the closed halfplane which is defined by the edge and contains all the points of  $P$  in a sufficiently small neighborhood of the edge’s midpoint, then the kernel of  $P$  is equal to the intersection of the inner halfplanes of all the edges of  $P$  and is therefore convex.

We will follow the terminology of Icking and Klein [5]; we briefly summarize it in this paragraph. From its starting position  $s$ , the robot probably does not see parts of the polygon  $P$  in which it stands; if the robot sees all of  $P$ , then  $s$  belongs to the kernel and the robot need not move. The hidden portions of the polygon are called *caves*. Each cave is adjacent to a reflex vertex of  $P$ , whose very existence creates the cave; these reflex vertices are called *constraint vertices* (Figure 1). A cave (associated with a constraint vertex  $v$ ) is characterized as either *left* if it lies to the left of the directed line  $\vec{sv}$ , or *right* otherwise. By extension, we say that a vertex is a *left constraint vertex* if it is a constraint vertex associated with a left cave, and similarly for a *right constraint vertex*. In Figure 1, the vertices  $v$  and  $w$  are left constraint vertices, and the shaded regions next to them are the associated caves; the vertex  $u$  is a right constraint vertex. For each of the constraint vertices  $v$ , we define its *inner halfplane* with respect to the current position  $p$  as the closed halfplane which is delimited by the line  $\overline{pv}$  and does not contain the corresponding cave.

From its starting position, the robot may detect zero or more left caves and zero or more right caves. If the robot sees at least one left cave, the following lemma holds (see [11] for a proof).

**Lemma 1.** *Suppose that from its starting position  $s$  in a simple polygon  $P$  the robot detects one or more left caves next to the constraint vertices  $l_1, \dots, l_k$  ( $k \geq 1$ ). Suppose further that no left constraint vertex exists such that the closure of the complement of its inner halfplane contains all the left constraint vertices. Then, the kernel of  $P$  is empty.*

A similar lemma holds for the right constraint vertices. Therefore, if the conditions of Lemma 1 hold, we need do nothing, since the polygon has empty kernel. Otherwise, there is a left constraint vertex such that the closure of the complement of its inner halfplane contains all the left constraint vertices and it is unique (if there are more than one vertices collinear with  $s$  then we choose the one farthest away from  $s$ ); we call this vertex *maximal left constraint vertex*. In Figure 1,  $v$  is the maximal left constraint vertex. In a similar fashion, we have

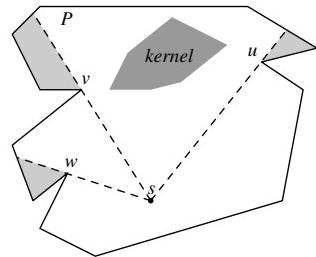


Figure 1

the *maximal right constraint vertex*. It can be proven that in a polygon with nonempty kernel, the left and right constraint vertices are not “intermixed” and this is why in papers on this problem which assume polygons with nonempty kernel, figures show the left and the right constraint vertices all gathered on the left and on the right of the polygon boundary respectively.

Crucial in the analysis of our strategy is the notion of a *curve with increasing chords*; a curve has increasing chords if  $|ad| \geq |bc|$  for any four points  $a, b, c, d$  lying on the curve in that order ( $|pq|$  denotes the length of the line segment connecting  $p$  and  $q$ ). For a plane curve with increasing chords, Rote proved that

**Lemma 2.** [13] *The length of a plane curve with increasing chords connecting two points  $a$  and  $b$  does not exceed  $\frac{2\pi}{3}$  times the length of the line segment connecting  $a$  and  $b$ .*

We close this section with a well known geometric fact and another lemma.

**Fact 1.** *Consider a circle with diameter  $ab$ . Then, the angle  $\widehat{apb}$  of the triangle with vertices  $a$ ,  $b$ , and  $p$  is less than, equal to, or greater than  $\pi/2$  if  $p$  lies outside, on the boundary, or inside the circle, respectively.*

**Lemma 3.** *Let  $C_1$  be a connected non-self-intersecting curve which does not intersect the line segment connecting its endpoints  $a$  and  $b$ , and  $C_2$  a convex polygonal line with the same endpoints which lies in the region enclosed by  $C_1$  and the line segment  $ab$ . Then, the length of  $C_2$  does not exceed the length of  $C_1$ .*

**Angle Notation:** Since three points define two angles (which sum up to  $2\pi$ ), in the following, the notation  $\widehat{abc}$  (where  $a, b, c$  are three non-collinear points) is meant to indicate the smallest of the two corresponding angles.

### 3 The Strategy

The basic motivation behind our strategy stems from the study of the simplest case, i.e., a single reflex vertex  $v$  whose incident edges are not both visible from the starting position  $s$ . Since the robot does not know the direction of the invisible edge  $e$  incident upon  $v$ , it does not know where the closest point  $t$  of the kernel might be. However, in all cases,  $t$  belongs to the semicircle with diameter  $sv$ , assuming that the semicircle lies in the polygon  $P$  (Figure 2). So, it seems a good idea to follow this semicircle.

Our strategy is based on this idea. Thus, the path of the robot consists of circular arcs and line segments; each circular arc belongs to a circle with diameter  $sp$ , where  $s$  is the starting position and  $p$  is a constraint vertex. This strategy makes the robot reach the kernel at its closest point to  $s$ .<sup>1</sup>

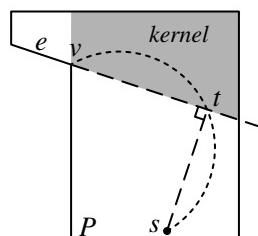


Figure 2

<sup>1</sup> It must be noted that this strategy is not optimal for the simple case of a single reflex vertex; it yields a worst-case competitive factor of  $\pi/2 \simeq 1.57$ . See [6], for a proof that the optimal competitive factor is  $\sim 1.212$ , and for a strategy achieving it.

We first consider the one-sided case, where there are only left or only right caves; our strategy for the general case consists of applying the one-sided case strategy twice, first for the left caves until we see them all, and then for the right caves (if needed).

### 3.1 The One-Sided Case

Without loss of generality, we consider the case where there are only left caves (the case where we have only right caves is similar). Until the robot sees all the left caves, there exist left constraint vertices and among them a maximal left constraint vertex, which may change as the robot moves. Initially, the robot finds the maximal left constraint vertex  $v_0$  as seen from the starting position  $s$  and starts following the semicircle with diameter  $sv_0$ . The two fundamental cases that characterize the robot's path are:

1. *A new maximal constraint vertex  $u$  is discovered.* Then, the robot will start following the semicircle with diameter  $su$  (Figure 3: point *a*). Interestingly, the current location of the robot belongs to both semicircles.
2. *The cave next to the currently maximal constraint vertex  $u$  becomes visible.* This implies that the second edge  $e$  incident upon  $u$  has become visible as well. Then, the robot at its current position, say,  $b$ , finds the new maximal constraint vertex. If no such vertex exists, then the entire polygon is visible and the robot has achieved its goal. If such a vertex exists —let it be  $v$ — and  $v$  is a constraint vertex just seen for the first time (for example, if  $v$  is the other endpoint of  $e$ ), then we execute the previous case. The remaining possibility is if  $v$  is a constraint vertex that has already been seen, in which case the robot walks along the line segment  $bu$  trying to reach (if possible) the semicircle with diameter  $sv$  (Figure 3: points *b* and *c*).

Note that it may be the case that the robot has to reach the currently maximal constraint vertex  $u$  in order to see the cave next to  $u$ . (This can only happen if  $u$  is the maximal left constraint vertex  $v_0$  seen from  $s$ .) In this case, if there exists a new maximal constraint vertex  $w$ ,  $w$  has to be a constraint vertex just discovered, for otherwise the polygon has empty kernel. The robot at  $u$  lies on or outside the semicircle with diameter  $sw$ , and it will try to walk along the line  $su$  away from  $s$  in an attempt to see the cave next to  $w$ .

The above two cases do not take into account the fact that the robot may take advantage of what it has seen. Clearly, the kernel of the polygon  $P$  is a subset of the inner halfplanes of the edges of  $P$  and of the inner halfplanes of the constraint vertices. Since the robot seeks to locate the kernel, it seems reasonable that it should not leave the inner halfplane of any of the polygon edges or constraint vertices which it sees or has seen. To be able to do that, the robot maintains the *free polygon* which is the subset of  $P$  in which the robot may walk. Initially, the free polygon is the intersection of the inner halfplanes of the visible edges and the visible constraint vertices from the starting point  $s$ . As

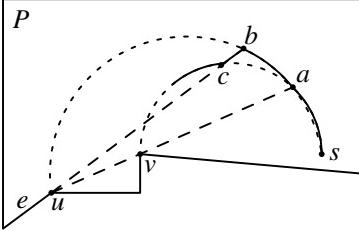


Figure 3

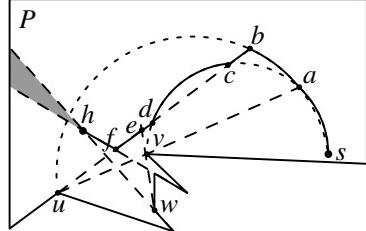


Figure 4

a new edge or a new constraint vertex becomes visible, the robot updates its free polygon by intersecting it with the corresponding inner halfplane. By requiring that the robot maintains the free polygon up to date and remains in it, we ensure that the portion of the polygon seen by the robot never decreases; at the same time, the free polygon keeps shrinking and when the robot reaches the kernel, the free polygon is precisely the kernel of  $P$ . Additionally, a left (resp., right) constraint vertex will remain so until both its incident edges become visible; it will not turn into a right (resp., left) constraint vertex, which might happen if the robot zig-zagged inside  $P$ .

At any time during its trip, the robot lies at a point, say,  $p$ , on the boundary of the current free polygon and it can only walk in the free polygon, that is, in the wedge delimited by the lines supporting the free polygon edges that are incident upon  $p$ . Since the free polygon is defined as the intersection of halfplanes, the opening angle of this wedge does not exceed  $\pi$ . Because the line supporting the edge to the left of  $p$  (with respect to the robot's motion towards the interior of the free polygon) bounds the current free polygon from the left, we call it a *left-bounding* line; similarly, the line supporting the edge to the right of  $p$  is a *right-bounding* line.

The following two cases complete the path planning strategy of the robot.

3. *The robot's intended course leads or lies outside the free polygon.* Then the robot walks along the boundary of the free polygon as close to the intended course as possible. In terms of left- and right-bounding lines, the robot walks along the left-bounding (right-bounding, respectively) line of the current free polygon if and only if the intended course leads to the left (right, respectively) of the free polygon.
4. *An edge that was not visible becomes visible.* Then, the robot updates the free polygon by intersecting it with the inner halfplane of that edge. Note that this case has to be executed in case 2.

An example is shown in Figure 4. It is important to note that the ending point  $h$  lies on the line supporting the edge which was seen last. Another important observation pertains to the way the value of the angle  $\widehat{psv_0}$  behaves, where  $p$  denotes the current position of the robot on its way from  $s$  to  $h$ , and  $v_0$  is the maximal left constraint vertex as observed from  $s$ . In the most general case, the following behavior of the angle  $\widehat{psv_0}$  is exhibited: it is initially  $\pi/2$ , then it

decreases, potentially reaching 0 but not decreasing below 0 (sub-path from  $s$  to  $f$  in Figure 4), and then it increases (sub-path from  $f$  to  $h$ ). (Note that the robot may walk along  $sv_0$ .) However, two special cases may arise: first, the value of  $\widehat{psv}_0$  is always decreasing from  $s$  to  $h$  (for example, consider the case that the caves of both  $v$  and  $w$  of Figure 4 were visible at  $f$ ), and second, the value of  $\widehat{psv}_0$  is always non-decreasing. The latter case may occur if, due to clipping, the left-bounding line of the free polygon is farther to the right from the semicircle with diameter  $sv_0$ ; in this case, the robot will not follow any of the semicircles defined by  $s$  and the maximal left constraint vertices.

**Lemma 4.** *Suppose that the angle  $\widehat{psv}_0$  decreases and then increases, reaching its minimum value when the robot is at the point  $x$ . Then,*

- (i)  *$x$  is either on or outside the corresponding semicircle,*
- (ii) *the part of the robot's path past  $x$  lies outside the semicircle defined by  $s$  and the currently maximal constraint vertex.*

### 3.2 The General Case

Our strategy for the general case consists of applying the one-sided strategy twice, first for the left caves and then for the right caves. Suppose that the robot is at point  $h$ , when it finally sees all the left caves. Then, the robot finds the maximal right constraint vertex  $u$  and updates its free polygon by intersecting it with the inner halfplane of  $u$  at  $h$ . The robot's intention is to walk along the semicircle  $C_{su}$  with diameter  $su$ ; however, it has to reach  $C_{su}$  first. To do this, the robot tries to walk along the line  $hu$  towards the semicircle; by walking in this direction, the robot does neither gain nor lose visibility of the cave next to  $u$ . Of course, this course is subject to clipping about the free polygon; so, if the path along  $hu$  towards  $C_{su}$  leads outside the free polygon, the robot follows left-bounding lines if  $h$  is inside  $C_{su}$  and right-bounding lines if  $h$  is outside  $C_{su}$ .

The final path consists of two sub-paths, one from  $s$  to  $h$  and the other from  $h$  to the final point  $t$ , each similar to the path shown in Figure 4. That is, each one of them consists of a number of clipped circular arcs and line segments (cases 1 and 2 of Section 3.1), potentially followed by one or more line segments that result from clipping whenever the corresponding semicircles fall outside the free polygon (Figures 5–7 show examples of paths). Our observation in Section 3.1 about the behavior of the values of the angle  $\widehat{psv}_0$  (where  $p$  is the robot's current position and  $v_0$  is the maximal left constraint vertex as observed from  $s$ ) is extended and implies that, in the most general case,  $\widehat{psv}_0$  is initially  $\pi/2$ , then decreases, potentially reaching 0 but not decreasing below 0, then it starts increasing assuming values up to  $\widehat{u_0sv_0}$  (where  $u_0$  is the maximal right constraint vertex as observed from  $s$ ), and then it may start decreasing again up to 0.

### 3.3 Simulating the Strategy

The obvious way to simulate a motion strategy involves starting at the predetermined starting position and executing small steps applying the rules of the

strategy. This method has the obvious disadvantage that a good approximation of the robot's path requires a large number of steps which may lead to increased execution time and large errors resulting from accumulated numerical errors at each step.

A second approach is to split the given polygon  $P$  into regions in each of which the robot follows the same curve. Clearly, we will have to split  $P$  about the lines supporting the polygon edges incident upon reflex vertices. Moreover, we need to split  $P$  about lines that connect pairs of (left or right) constraint vertices that consecutively become maximal. To do that, we find the tree of shortest paths inside  $P$  from  $s$  to all the reflex vertices and we split  $P$  about the lines supporting the edges of this tree as well. Then, the robot can traverse any of the resulting regions in one computational step; the only computation in each region involves finding the points of intersection of the path with the region boundary. This method involves fewer steps compared to the previous one but it requires computing the partition of the polygon about the above mentioned lines; the total number of these lines is linear in the number  $n$  of polygon vertices. Building the partition requires  $O(n^2)$  space and it can be done incrementally in  $O(n^2)$  time in a fashion similar to the incremental construction of an arrangement of lines; see [3] and [2]. The free polygon is maintained by turning on or off a bit associated with each region.

### 3.4 Path Properties

It is interesting to observe that every point of the robot's path belongs either to a semicircle defined by the starting point and a vertex of the polygon  $P$  (a maximal constraint vertex) or to the line supporting an edge of  $P$ . This guarantees that an actual robot following our strategy is not expected to deviate from the intended course, as opposed to other strategies where this is possible because the motion of the robot is dependent on the current position. For example, in Icking and Klein's strategy, the robot follows the bisector of an angle with apex the current position; but then, due to accumulated numerical error, the robot may deviate substantially from the expected course.

Additionally, the following lemmata establish two important properties of the robot's path (proofs can be found in [11]).

**Lemma 5.** *The path resulting from the application of the above described strategy reaches the kernel of the polygon at the kernel's point that is closest to the starting point  $s$ .*

**Lemma 6.** *The path that the robot follows in accordance with our strategy consists of  $O(n)$  line segments or circular arcs, where  $n$  is the number of vertices of the polygon  $P$ .*

## 4 Competitiveness Analysis

In order to compute the competitive factor of our strategy, we need to compute the worst-case ratio of the length of the path resulting from the application of our strategy over the length of the line segment connecting the starting point  $s$  to the ending point  $t$ . Obviously, the worst case scenario involves double application of the one-sided case. Our analysis relies on computing the competitive factor of an “augmented” path (we ignore (most of) the clipping) whose length is no less than the length of the actual path traveled.

Before we describe the “augmentation” procedure, we review the important stops in the robot’s path and define the l-path and r-path which will be used to augment the path. The robot first applies the one-sided strategy trying to see all the left caves; let  $h$  be the final point during this phase, that is, the point from where all the left caves are visible. Then, the robot applies the one-sided strategy again, for the right caves this time. As mentioned in Section 3.2, the angle  $\widehat{psv_0}$  (defined by the current position  $p$  of the robot, the starting position  $s$ , and the maximal left constraint vertex  $v_0$  observed from  $s$ ) decreases, then it may increase and finally it may decrease again; let  $x$  and  $y$  be the turning points where these changes of monotonicity occur (if the robot walks along the line  $sx$  or  $sy$ , we let  $x$  and  $y$  be the closest such points to  $s$ ). Note that  $x$  may coincide with  $h$  or may be before or after  $h$  along the robot’s path;  $y$  may coincide with  $t$ , although this is not true in the most general case. Moreover, as mentioned earlier, the point  $h$  lies on the line supporting the polygon edge that just became visible at  $h$ ; let  $l_h$  be that line. Then,  $l_h$  is a right-bounding line of the free polygon at  $h$ . Similarly, the ending point  $t$  lies on the line  $l_t$  supporting the edge that became visible last, and  $l_t$  is a left-bounding line of the free polygon at  $t$ .

We define the *l-path* as the path that the robot would follow if it only applied cases 1 and 2 of Section 3.1 from its starting position  $s$  until it either saw all the left caves or reached the line  $sx$ , whichever came first; in the former case, we extend the l-path by adding a line segment along the left-bounding line of the free polygon from the l-path’s final point to the point of intersection with  $sx$ . Because clipping is ignored, this left-bounding line supports a polygon edge next to a maximal left constraint vertex; this edge is not necessarily the edge that became visible last. As a summary, the l-path consists of a sequence of circular arcs (arcs  $sa$ ,  $bc$  of Figure 3) occasionally separated by a line segment along a line supporting an initially invisible polygon edge (segment  $bc$  of Figure 3). We define the *r-path* similarly: this is the path that the robot would follow if it only applied cases 1 and 2 of Section 3.1 starting from  $s$  until it either saw all the right caves or reached the line  $sy$ ; again, if the robot has seen all the right caves before it reached the line  $sy$ , we extend the r-path accordingly. We finally define the *l-region* as the closed region bounded by the l-path and the line  $sx$ ; similarly, the *r-region* is the closed region bounded by the r-path and the line  $sy$ . We note that:

**Observation 1.** *The point  $h$  from which all the left caves are finally visible does not belong to the interior of the l-region. Similarly, the final point  $t$  from which the entire polygon is visible does not belong to the interior of the r-region.*

The robot tries to follow the l-path and the r-path if possible, or otherwise stay as close to them as possible. On its course from the starting point  $s$  to  $h$  (the case is similar for the part from  $h$  to the ending point  $t$ ), it follows (parts of) the l-path, may move outside the l-region due to clipping about a left-bounding line (when the l-path leads farther left than the left boundary of the free polygon), or may move inside the l-region due to clipping about a right-bounding line (when the l-path leads farther right than the right boundary of the free polygon). In general, the robot may move in and out of the l-region several times; after it has moved in, it may walk along several different right-bounding lines (tracing a convex curve inside the l-region), whereas after it has moved out, it may follow several different left-bounding lines (tracing a concave curve outside the l-region). It is important to observe:

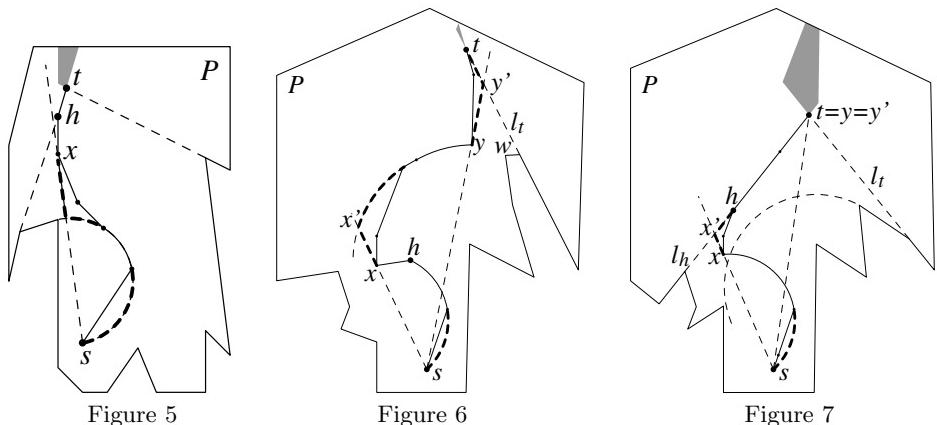
**Observation 2.** *The robot never follows a left-bounding line right after a right-bounding line (or vice versa) except at the point  $h$  where it sees all the left caves.*

The observation follows from the fact that the robot tries to stay as close to the corresponding semicircle as it can and if this is farther left (right, respectively) than the left (right, respectively) boundary of the free polygon, the robot will keep following the left (right, respectively) boundary of the free polygon until it reaches it, if ever.

#### 4.1 Augmenting the Robot's Path

Now we are ready to see how the actual robot's path is being augmented; we will also define the points  $x'$  and  $y'$  which will be crucial in partitioning the augmented path into curves with increasing chords. We concentrate on the most general case in which  $x \neq s$  (i.e., the angle  $\widehat{psv_0}$  starts by decreasing) and  $x \neq t$ ; the special cases where  $x = s$  or  $x = t$  yield smaller competitive factors (see [II]). Note that  $y$  may or may not coincide with  $t$ .

1. *the part of the robot's path from  $s$  to  $x$ :* We recall that  $x$  may be either on the l-path or outside the l-region; in the latter case and if additionally  $h$  coincides with or is reached after  $x$ , then the robot has been walking along left-bounding lines from the last point of its course on the l-path up to  $x$ . Recall also that  $h$  is either on the l-path or outside the l-region (Observation 1); if it is outside the l-region, then again the robot has been walking along left-bounding lines. In all cases where the robot walks along left-bounding lines after it leaves the l-region (no matter whether  $h$  is reached before or after  $x$ ), the sub-path from  $s$  to  $x$  is augmented by considering the entire l-path, followed by a line segment from the final point of the l-path to  $x$  along  $sx$  (Figure 5); this includes as a special case the case where  $x$  belongs to the l-path. It remains to consider the cases where the robot walks along right-bounding lines. There are two cases to consider: first,  $h$  belongs to the l-path,  $x$  is reached after  $h$ , and the robot walks along a right-bounding line past  $h$  towards  $x$ , and second,  $h$  is outside the l-region,  $x$  is reached after  $h$ , and the robot walks along a right-bounding line past  $h$  towards  $x$ ; both cases imply  $x = t$  and yield smaller competitive factors (see [II]).



2. *the part of the robot's path from  $x$  to  $y$ :* We distinguish two cases depending on whether the robot walks along left- or right-bounding lines past  $x$ .

(i) the robot walks along a left-bounding line past  $x$ . If the point  $h$  is before  $x$  or coincides with  $x$ , then  $x$  must belong to the r-region for the robot to follow a left-bounding line past  $x$ . We let  $x'$  be the point of intersection of  $sx$  with the r-path, and we augment the sub-path from  $x$  to  $y$  by considering the line segment  $xx'$ , followed by the r-path up to its intersection with the line  $sy$ , followed by the line segment from that point to  $y$  (Figure 6). If the point  $h$  is after  $x$ , then past  $h$  the robot may walk along a left- or a right-bounding line depending on whether  $h$  belongs to the r-region or not. Let  $q$  be the point of intersection of the lines  $sx$  and  $l_h$ . If  $h$  is outside the r-region, or if  $h$  belongs to the r-region but  $q$  does not, we set  $x' = q$  and we augment the path by considering the line segment  $xx'$  (along  $sx$ ), followed by a line segment along  $l_h$  from  $x'$  to the point of intersection with the r-path, followed by a line segment from that point to  $y$  along  $sy$  (Figure 7). If both  $q$  and  $h$  belong to the r-region, then we let  $x'$  be the point of intersection of the line  $sx$  with the r-path, and the sub-path from  $x$  to  $y$  is augmented by considering the line segment  $xx'$  (along  $sx$ ), followed by the r-path from  $x'$  to its final point on the line  $sy$ , followed by the line segment from that point to  $y$  (along  $sy$ ); the situation is similar to the one depicted in Figure 6.

(ii) the robot walks along a right-bounding line past  $x$ . Then,  $h$  cannot be before  $x$ , for, if  $h$  were reached before  $x$ , the robot must have been walking along right-bounding lines from  $h$  to  $x$ ; this implies that  $x = t$ , a contradiction to the continuation of the path past  $x$ . Moreover,  $h$  cannot be after  $x$  either; if  $h$  were reached after  $x$ , then  $h$  would be outside the l-region and the robot would be walking along left-bounding lines from  $x$  to  $h$ . Therefore,  $h = x$ , and we set  $x' = h$ . Additionally,  $h$  lies outside the r-region (otherwise, the robot would not be following a right-bounding line past  $x$ ). Let  $q$  be the point of intersection of  $l_h$  with the r-path (if  $l_h$  intersects a line segment of the r-path, then  $q$  is the point of intersection of  $l_h$  with the immediately

following semicircle); if the line  $l_h$  does not intersect the r-path, we let  $q$  be the point of intersection of  $l_h$  and  $sy$ . Then, the sub-path from  $x$  to  $y$  is augmented by considering the line segment  $xq$  along  $l_h$ , potentially followed by the r-path from  $q$  to its intersection with the line  $sy$  (if  $q$  does not belong to the line  $sy$ ), followed by the line segment from that point to  $y$ .

3. *the part of the robot's path from  $y$  to the final point  $t$ :* If  $y = t$ , then we set  $y' = y = t$ . If  $y \neq t$ , then the path past  $y$  lies outside the corresponding semicircles (Lemma 4) and the robot on its way to  $t$  walks along right-bounding lines only. So, this part of the actual path is augmented by considering the polygonal line formed by the segments  $yy'$  and  $y't$ , where  $y'$  is the point of intersection of the lines  $sy$  and  $l_t$  (Figure 6).

It is important to observe that the augmented path does not cross itself. Moreover, the augmented path proceeds along or to the left of the left-bounding lines that the robot follows, and along or to the right of the right-bounding lines, thus enclosing the actual robot's path. Therefore, we have:

**Observation 3.** *The path traveled by the robot and the augmented path have the same endpoints.*

**Observation 4.** *The path traveled by the robot can be produced by clipping the augmented path about the edges of a (shrinking) convex polygon.*

## 4.2 The Competitive Factor

With respect to the points  $x'$  and  $y'$ , the augmented path can be seen as the concatenation of three sub-paths, one from  $s$  to  $x'$ , one from  $x'$  to  $y'$ , and one from  $y'$  to the final point  $t$ . The sub-path from  $s$  to  $x'$  consists of circular arcs occasionally separated by a line segment along a line supporting an initially invisible polygon edge (cases 1 and 2 of Section 3.1), potentially ending with a line segment along the line  $sx$ . The sub-path from  $x'$  to  $y'$  consists mainly of arcs and line segments (in accordance with cases 1 and 2 of Section 3.1) as well, but may begin with a line segment along  $l_h$ , and may end with a line segment along the line  $sy$ ; the sub-path may degenerate into a two-segment polygonal line, one along  $l_h$  and the other along  $sy$ . Finally, the sub-path from  $y'$  to  $t$  is simply a line segment. See Figures 6-7. More importantly, the following lemmata hold.

**Lemma 7.** *The (counterclockwise) angle  $\widehat{sx'y'}$  is at least equal to  $\pi/2$ .*

*Proof.* The definition of the point  $x'$  in the case 2 of the preceding section suggests that we need to consider two cases. First, suppose  $x'$  is the point of intersection of  $sx$  with the r-path (if  $x$  is inside or on the boundary of the r-region). Then,  $x'$  lies on the semicircle of the currently maximal right constraint vertex (case 1 of Section 3.1), or on the line supporting an edge incident upon a right-constraint vertex which was initially invisible and became visible (case 2 of Section 3.1); in the latter case,  $x'$  lies inside the semicircle associated with the right constraint vertex. In either case, if  $w$  is the right constraint vertex, then

the angle  $\widehat{sx'w}$  is at least equal to  $\pi/2$ . Moreover, the point  $y$  and (a fortiori) the point  $y'$  lie on or to the left of the directed line  $\overrightarrow{x'w}$  (see Figure 6). Therefore,  $\widehat{sx'y'} \geq \widehat{sx'w}$ , and the lemma follows.

Suppose now that  $x'$  is the point of intersection of  $sx$  with  $l_h$ : this case occurs if  $h$  is reached after  $x$ , or  $x' = h$  and  $h$  lies outside the r-region. In either case,  $x'$  coincides with or is farther away from  $s$  than  $x$ ; since  $x$  is on the boundary or outside the l-region (Lemma 4),  $x'$  lies on or outside the semicircle of the last maximal left-constraint vertex, say,  $v$ . Then,  $\widehat{sx'v} \leq \pi/2$  (Fact 1). The lemma follows from the fact that  $y$  and (a fortiori)  $y'$  belong to the inner halfplane of  $l_h$  and thus the angle  $\widehat{sx'y'}$  is at least equal to  $\pi - \widehat{sx'v}$  (see Figure 7). ■

Similarly,

**Lemma 8.** *If  $y \neq t$ , the (clockwise) angle  $\widehat{sy't}$  is at least equal to  $\pi/2$ .*

**Lemma 9.** *The sub-path of the augmented path from  $s$  to  $x'$  is a curve with increasing chords.*

*Proof.* For any point  $p$  (other than  $s$ ), we define the quadrants  $A_p$ ,  $B_p$ ,  $C_p$  and  $D_p$  at  $p$  as the four closed quadrants determined by the line  $sp$  and its perpendicular at  $p$ : the quadrant  $A_p$  is the quadrant that contains  $s$  and lies to the right of the directed line  $\overrightarrow{sp}$ , while the other quadrants  $B_p$ ,  $C_p$  and  $D_p$  follow quadrant  $A_p$  in counterclockwise order around  $p$ . We first prove that for any point  $p$  of this sub-path, the part of the augmented path from  $s$  to  $p$  belongs to the closed quadrant  $A_p$  of  $p$ , while the part of the path from  $p$  to  $x'$  belongs to the closed quadrant  $C_p$  of  $p$ . One needs to consider the different cases for  $p$ : on a circular arc, at the intersection of two arcs, at the intersection of an arc and a line segment, on a line segment. This follows from the fact that for any point  $q$  of a semicircle with diameter  $ab$ , the angle  $\widehat{aqb}$  is equal to  $\pi/2$  (see Fact 1). (Figure 8 gives some examples for illustration purposes; the crosses indicate the lines delimiting the quadrants.) Next, we consider 4 points  $a$ ,  $b$ ,  $c$  and  $d$  in that order along the augmented path. We draw the corresponding quadrants for the points  $b$  and  $c$  and draw the two lines  $l_b$  and  $l_c$  perpendicular to  $bc$  that pass by  $b$  and  $c$  respectively (Figure 9). Since  $c$  belongs to the quadrant  $C_b$  of  $b$ ,  $l_b$  lies in the closure of the wedge defined by the quadrants  $B_b$  and  $D_b$  of  $b$ . Similarly, since  $b$  belongs to the quadrant  $A_c$  of  $c$ ,  $l_c$  lies in the closure of the wedge defined by the quadrants  $B_c$  and  $D_c$  of  $c$ . Moreover, the point  $a$  lies in the quadrant  $A_b$  of  $b$ , that is, to the left of  $l_b$ . Similarly, the point  $d$  lies in the quadrant  $C_c$  of  $c$ , that is, to the right of  $l_c$ . Therefore, the length of  $ad$  is no less than the perpendicular distance of  $l_b$  and  $l_c$ , which by construction is equal to  $bc$ . ■

In a similar fashion, although with a little more effort because the sub-path of the augmented path between  $x'$  and  $y'$  may begin with a line segment, we can prove:

**Lemma 10.** *The sub-path of the augmented path from  $x'$  to  $y'$  is a curve with increasing chords.*

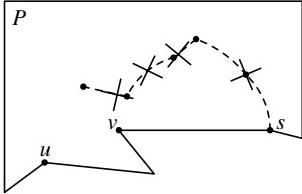


Figure 8

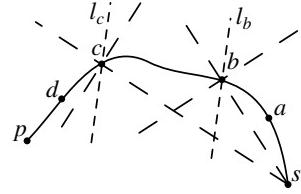


Figure 9

From the above, we conclude

**Theorem 1.** *Our strategy has a competitive factor of  $\sqrt{2(2\pi/3)^2 + 1} \simeq 3.126$ .*

*Proof.* Clearly, the length of the actual path traveled by the robot is no more than the length of the augmented path, as clipping with convex polygonal lines or curves leads to reduced path length (Observation 4 and Lemma 3). So, an upper bound on the ratio of the length of the augmented path over the length of the line segment  $st$  readily implies an upper bound on the competitive factor that we seek. Figure 10 shows the skeleton of the augmented path in the worst case; the angles  $\alpha = \widehat{sx'y'}$  and  $\beta = \widehat{sy't}$  are at least equal to  $\pi/2$  (Lemmata 7 and 8). Let us denote by  $|\tilde{pq}|$  the length of the path from  $p$  to  $q$  as opposed to  $|pq|$  which denotes the length of the line segment  $pq$ . Then the competitive factor  $r$  is

$$r = \frac{|\tilde{sx'}| + |\tilde{x'y'}| + |y't|}{|st|} \leq \frac{\frac{2\pi}{3}|sx'| + \frac{2\pi}{3}|x'y'| + |y't|}{|st|},$$

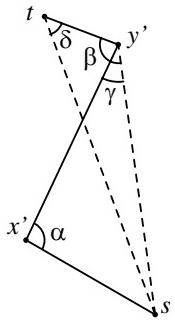


Figure 10

since the augmented sub-paths from  $s$  to  $x'$  and from  $x'$  to  $y'$  are curves with increasing chords (Lemmata 9 and 10) and therefore their lengths are not more than  $2\pi/3$  times the lengths of the line segments  $sx'$  and  $x'y'$  respectively (Lemma 2). If we apply the law of sines in the triangles  $sx'y'$  and  $sy't$ , factor out the length  $|sy'|$ , and maximize using partial derivatives, we find

$$r \leq \frac{\frac{2\pi}{3} \frac{\sin\gamma + \sin(\alpha+\gamma)}{\sin\alpha} + \frac{\sin(\beta+\delta)}{\sin\delta}}{\frac{\sin\beta}{\sin\delta}} \leq \frac{\frac{2\pi}{3} \sqrt{2} + \frac{\sin(\beta+\delta)}{\sin\delta}}{\frac{\sin\beta}{\sin\delta}} \leq \sqrt{2(2\pi/3)^2 + 1}.$$

where  $\pi/2 \leq \alpha < \pi$ ,  $\pi/2 \leq \beta < \pi$ ,  $0 < \gamma < \pi - \alpha$  and  $0 < \delta < \pi - \beta$ : the term  $\frac{\sin\gamma + \sin(\alpha+\gamma)}{\sin\alpha}$  is decreasing as  $\alpha$  increases and is thus maximized for  $\alpha = \pi/2$  and  $\gamma = \pi/4$ ; similarly, the overall fraction is maximized for  $\beta = \pi/2$ . ■

## 5 Concluding Remarks – Open Problems

We presented a strategy which enables a point robot to reach the point  $t$  of the kernel that is closest to the starting point  $s$ , and guarantees that the length of

the path traveled is not longer than 3.126 times the length of the line segment  $st$  (that is, 3.126 times the shortest possible off-line path). Our strategy has the interesting feature that the robot reaches the kernel at precisely the closest point  $t$ . We note that the above competitive factor cannot be guaranteed when the polygon has empty kernel (in such cases, the competitive factor is defined as the ratio of the length of the path that a strategy imposes over the length of the shortest path which establishes that the kernel is empty), and this holds for all strategies where a point of the polygon seen by the robot never ceases to be in the robot's visible region thereafter (enforced by means of the free polygon in this work, and by means of the gaining and keeping wedges in [5] and [9]).

Experimental results seem to suggest that the actual competitive factor is smaller than the theoretical competitive factor of 3.126. If true, it would be interesting to come up with tighter theoretical bounds on the competitive factor of our strategy. Of course, the ultimate open question is to invent strategies with smaller competitive factors which will close the gap between the current upper bound of  $\sim 3.126$  and the lower bound of  $\sim 1.48$ . To this effect, perhaps ideas like the ones in [6] may be of help.

Finally, better competitive solutions are needed for other motion planning problems in unknown environments. López-Ortiz and Schuierer [10] have addressed two interesting problems in this class: finding out whether a given polygon is star-shaped (i.e., it has non-empty kernel), and locating a target (to be recognized when seen) in a polygon with non-empty kernel. The currently best competitive factor for the first problem is 46.35. The currently best competitive factor for the second problem is 12.72 and is coupled with a lower bound of 9.

## Acknowledgements

I would like to thank Vassilios Karaiskos whose program (implementing our strategy) helped to produce several of the included figures.

## References

1. A. Blum, P. Raghavan, and B. Schieber, “Navigating in unfamiliar Geometric Terrain,” *Proc. 23th ACM Symposium on Theory of Computing* (1991), 494–504.
2. H. Edelsbrunner and L.J. Guibas, “Topologically sweeping an arrangement,” *Journal of Computer and Systems Science* 38 (1989), 165–194. Corrigendum in 42 (1991), 249–251.
3. H. Edelsbrunner, J. O'Rourke, and R. Seidel, “Constructing arrangements of lines and hyperplanes with applications,” *SIAM Journal on Computing* 15(2) (1986), 341–363.
4. Y.K. Hwang and N. Ahuja, “Gross Motion Planning – A Survey,” *ACM Computing Surveys*, Vol. 24, No. 3 (1992), 219–291.
5. C. Icking and R. Klein, “Searching for the Kernel of a Polygon — A Competitive Strategy,” *Proc. 11th ACM Symp. on Computational Geometry* (1995), 258–266.
6. C. Icking, R. Klein, and L. Ma, “An Optimal Competitive Strategy for Looking around a Corner,” *Proc. 5th Canadian Conference on Computational Geometry* (1993), 443–448.

7. C. Icking, A. Lopez-Ortiz, S. Schuierer, and I. Semrau, "Going Home through an Unknown Street," *Technical Report 228*, Dept of Computer Science, FernUniversität Hagen, Germany, 1998.
8. J.-H. Lee and K.-Y. Chwa, "Tight Analysis of a Self-Approaching Strategy for the online Kernel-Search Problem" *Information Processing Letters* 69 (1999), 1–52.
9. J.-H. Lee, C.-S. Shin, J.-H. Kim, S.Y. Shin, and K.-Y. Chwa, "New Competitive Strategies for Searching in Unknown Star-Shaped Polygons," *Proc. 13th ACM Symposium on Computational Geometry* (1997), 427–429.
10. A. López-Ortiz and S. Schuierer, "Position-independent near optimal Searching and on-line Recognition in Star Polygons," *Proc. 5th International Workshop on Algorithms and Data Structures – WADS* (1997), 284–296.
11. L. Palios, "A New Competitive Strategy for Reaching the Kernel of an Unknown Polygon," *Technical Report* (1999), Dept. of Computer Science, Univ. of Ioannina.
12. C. Papadimitriou and M. Yannakakis, "Shortest paths without a map," *Theoretical Computer Science* 84 (1991), 127–150.
13. G. Rote, "Curves with increasing chords," *Mathematical Proceedings of the Cambridge Philosophical Society* 115 (1994), 1–12.
14. D. Sleator and R.E. Tarjan, "Amortized efficiency of list update and paging rules," *Communications of the ACM* 28 (1985), 202–208.

# The Enhanced Double Digest Problem for DNA Physical Mapping

Ming-Yang Kao<sup>1</sup> \*, Jared Samet<sup>1</sup>, and Wing-Kin Sung<sup>2</sup> \*\*

<sup>1</sup> Department of Computer Science, Yale University, New Haven, CT 06520, USA  
`kao-ming-yang@cs.yale.edu, jared.samet@yale.edu`

<sup>2</sup> E-business Technology Institute, University of Hong Kong, Hong Kong  
`wksung@eti.hku.hk`

**Abstract.** The *double digest problem* is a common NP-hard approach to constructing physical maps of DNA sequences. This paper presents a new approach called the *enhanced double digest problem*. Although this new problem is also NP-hard, it can be solved in linear time under a certain restriction, which is satisfied reasonably frequently.

**Key words.** DNA physical mapping, fast algorithms, graph-theoretic techniques, NP-hardness

## 1 Introduction

The physical mapping of DNA is a key problem in computational biology [4]. A *map* of a DNA sequence consists of the locations of some given small sequences like e.g. GAATTC. Biologists use such maps in a preparatory step to determine the target DNA sequence [5].

A common technique of constructing maps uses restriction enzymes to cut a DNA sequence at the positions where a particular short DNA sequence appears. These positions are called *restriction sites*. One approach to modeling map construction is the *double digest* (DD) problem. Given two restriction enzymes  $\mathcal{A}$  and  $\mathcal{B}$ , this approach cuts a target DNA sequence using enzyme  $\mathcal{A}$ , enzyme  $\mathcal{B}$ , and both enzymes, separately. It is a biology fact that the restriction sites for enzymes  $\mathcal{A}$  and  $\mathcal{B}$  do not coincide. Throughout this paper, we make use of this fact. Let  $A$ ,  $B$  and  $C$  be the three multisets of the lengths of the fragments formed after applying enzyme  $\mathcal{A}$ , enzyme  $\mathcal{B}$  and both enzymes to the target DNA sequence, respectively. Given  $A$ ,  $B$  and  $C$ , the DD problem asks for permutations of the lengths in  $A$  and  $B$  such that if these sets of lengths are plotted on top of one another, the lengths of all the resulting subintervals formed due to overlapping match exactly the lengths in  $C$ . See Figure 1 for an example.

Many algorithms [6,7,8,10] have been proposed for the DD problem. Stefik [9] gave the first algorithm using artificial intelligence. Fitch, Smith and Ralph

\* Research supported in part by NSF Grant 9531028.

\*\* This work is developed at Yale University.

(a)	17	37	12	15	9
(b)	46		38		6
(c)	17	29	8	12	15
			3	6	

**Fig. 1.** Stripes (a), (b) and (c) show the fragments resulting from the applications of enzyme  $\mathcal{A}$ , enzyme  $\mathcal{B}$  and both enzymes, respectively. In strip (c), the subfragments are created due to the overlapping between fragments in (a) and those in (b).

[1] reduced the DD problem to the set partition problem. Goldstein and Waterman [3] approached this problem with a stochastic annealing heuristic for the traveling salesman problem. They also showed that the DD problem is NP-hard by reducing the set partition problem to it.

This paper suggests a new approach, called the *enhanced double digest* (EDD) problem. The EDD problem uses  $A, B, C$  and some additional length information; see Section 2 for the details of the approach. Although the EDD problem is still NP-hard, we show that if the lengths in  $C$  are all distinct, it can be solved in linear time. We also generalize the algorithm for the case where the number of duplicates in  $C$  is bounded by a constant. The time complexity of this generalized algorithm remains linear. Based on preliminary analysis, these constraints on duplicates in  $C$  can be satisfied with a reasonable probability.

Section 2 details the new approach to define the EDD problem formally. Section 3 gives the linear-time algorithm for the case where  $C$  is duplicate-free. Also, it generalizes the algorithm to handle a small number of duplicate lengths. Section 4 proves that the EDD problem is NP-hard. Section 5 concludes with some directions for further work.

## 2 Problem Formulation

Consider a target DNA sequence and two restriction enzymes  $\mathcal{A}$  and  $\mathcal{B}$ .

- By applying enzyme  $\mathcal{A}$  (respectively,  $\mathcal{B}$ ) to the target DNA sequence, we obtain  $p$  (respectively,  $q$ ) fragments. Let  $A = \{a_1, \dots, a_p\}$  (respectively,  $B = \{b_1, \dots, b_q\}$ ) be the multiset of the lengths of these  $p$  (respectively,  $q$ ) fragments.
- For  $i = 1, \dots, p$ , let  $\hat{a}_i$  be the fragment corresponding to  $a_i$ . We apply enzyme  $\mathcal{B}$  to the fragment  $\hat{a}_i$  and obtain a set of subfragments. Let  $AB_i$  be the multiset of the lengths of these subfragments.
- For  $j = 1, \dots, q$ , let  $\hat{b}_j$  be the fragment corresponding to  $b_j$ . We apply enzyme  $\mathcal{A}$  to the fragment  $\hat{b}_j$  and obtain a set of subfragments. Let  $BA_j$  be the multiset of the lengths of these subfragments.

For the example in Figure 1, the following length information is gathered:

- $A = \{a_1 = 9, a_2 = 12, a_3 = 15, a_4 = 17, a_5 = 37\}$ ;  
 $B = \{b_1 = 6, b_2 = 38, b_3 = 46\}$ ;
- $AB_1 = \{3, 6\}; AB_2 = \{12\}; AB_3 = \{15\}; AB_4 = \{17\}; AB_5 = \{8, 29\}$ ;
- $BA_1 = \{6\}; BA_2 = \{3, 8, 12, 15\}; BA_3 = \{17, 29\}$ .

It is easily verified that the data found in this way has the following properties:

**Fact 1.**

1. For  $i = 1, \dots, p$ ,  $a_i = \sum_{c \in AB_i} c$ . For  $j = 1, \dots, q$ ,  $b_j = \sum_{c \in BA_j} c$ .
2.  $\bigcup_i AB_i = \bigcup_j BA_j = C$ .
3.  $|C| = |A| + |B| - 1$ .

*Proof.* Straightforward.

Given  $A, B, AB_1, \dots, AB_p, BA_1, \dots, BA_q$ , the *enhanced double digest problem*  $\mathcal{P}$  asks for a *valid permutation*  $(\pi_A, \pi_B)$  of the elements in  $A$  and  $B$  such that the following can be achieved. When the fragments  $\hat{a}_i$  for  $a_i \in A$  and  $\hat{b}_j$  for  $b_j \in B$  are plotted on the same line according to the order given by  $\pi_A$  and  $\pi_B$ , a set of subfragments is formed due to overlapping. The multiset  $C$  of the lengths of these subfragments is required to be equal to  $\bigcup_{i=1}^p AB_i = \bigcup_{j=1}^q BA_j$ . In addition,

- for every  $a_i \in A$  (respectively,  $b_j \in B$ ),  $AB_i$  (respectively,  $BA_j$ ) is equal to the multiset of the lengths of the subfragments which overlap with  $\hat{a}_i$  (respectively,  $\hat{b}_j$ ).

Note that an instance of this problem may have no solution or more than one valid permutation. The algorithms given in Section 3 can recover all valid permutations, if any exists.

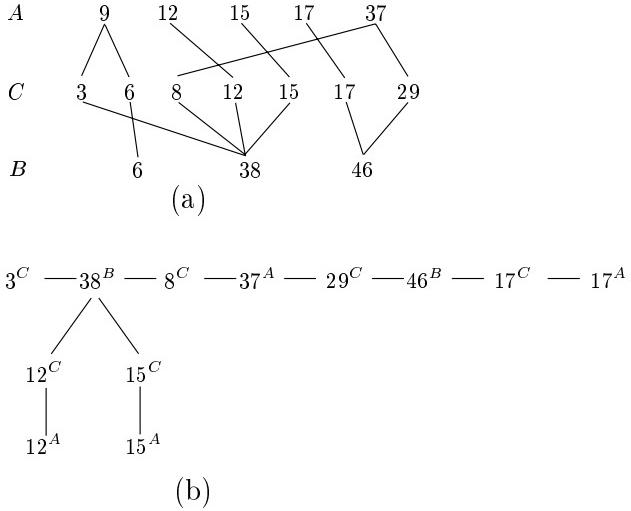
### 3 An Efficient Algorithm

Unless otherwise stated, this section assumes that  $C$  has no duplicates. Let  $n = |C|$ . This section shows that the EDD problem  $\mathcal{P}$  can be solved in  $O(n)$  time.

Section 3.1 formulates the EDD problem as a graph problem. Section 3.2 describes the linear-time algorithm. Section 3.3 discusses how to generalize this linear-time algorithm to the case where  $C$  may contain a small number of duplicates.

#### 3.1 A Graph Representation

Given  $A, B, AB_1, \dots, AB_p, BA_1, \dots, BA_q$ , we construct an undirected graph  $G$  as follows.



**Fig. 2.** The graph  $G$  in (a) is constructed from the example in Figure 1.  $G$  can be redrawn into a spanning tree as shown in (b). The superscript  $A, B$  or  $C$  of each node denotes whether the node belongs to  $A, B$  or  $C$ .

- The node set of  $G = A \cup B \cup C$ .
- For every  $a_i \in A$  and every  $x \in C$ ,  $(a_i, x) \in G$  if  $x \in AB_i$ .
- For every  $b_j \in B$  and every  $x \in C$ ,  $(b_j, x) \in G$  if  $x \in BA_j$ .

From the definition , we can observe that  $G$  satisfies the following lemma.

**Lemma 1.**  *$G$  is connected. For each node in  $A \cup B$ , its degree is at least 1 and it is adjacent to nodes in  $C$  only. Also, every node in  $C$  connects to exactly one node in  $A$  and one node in  $B$ .*

*Proof.* Straightforward based on the assumption that  $C$  has no duplicates.

If  $\mathcal{P}$  has a valid permutation,  $G$  has two more properties as stated in Lemma 2. Figure 2 illustrates an example. A *diameter* of a tree is a path with the largest number of edges. A *dangler* is a 2-node-long path. Given a tree  $T$ , a subtree  $\tau$  of  $T$  is said to be *hanged on* a path  $P$  in  $T$  if  $\tau$  is a tree in the spanning forest  $T - P$ .

**Lemma 2.** *If  $\mathcal{P}$  has a valid permutation, then the following statements hold.*

1.  *$G$  is a spanning tree.*
2. *For any diameter  $S$  of  $G$ , the subtrees hanged on  $S$  must be danglers.*

*Proof.*

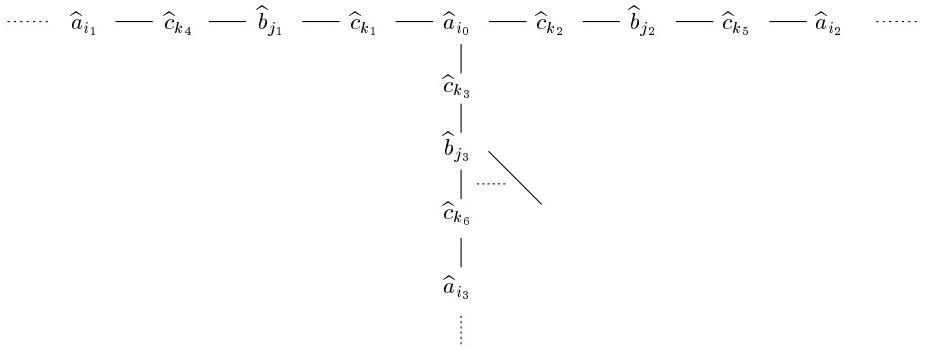
Statement 1 To prove by contradiction, suppose that  $G$  contains a cycle  $D$ . By the construction of  $G$ ,  $D$  must be of the form

$$a_{i_1}, c_{k_1}, b_{j_1}, c_{k_2}, a_{i_2}, c_{k_3}, b_{j_2}, c_{k_4}, \dots, c_{k_{2z}}, a_{i_{z+1}},$$

where  $i_1 = i_{z+1}$ ;  $a_{i_1}, \dots, a_{i_z} \in A$ ;  $b_{j_1}, \dots, b_{j_z} \in B$ ; and  $c_{k_1}, \dots, c_{k_{2z}} \in C$ .

By definition, if  $a_i, c_k, b_j$  is a path in  $G$ , then  $\hat{a}_i$  and  $\hat{b}_j$  overlap by  $\hat{c}_k$  in any valid permutation of  $\mathcal{P}$ . Thus, for  $1 \leq \ell \leq z - 1$ , the existence of the subpath  $a_{i_\ell}, \dots, a_{i_{\ell+2}}$  of  $D$  in  $G$  means that  $\hat{b}_{i_\ell}$  overlaps with  $\hat{a}_{i_\ell}$  and  $\hat{a}_{i_{\ell+1}}$  and  $\hat{b}_{i_{\ell+1}}$  overlaps with  $\hat{a}_{i_{\ell+1}}$  and  $\hat{a}_{i_{\ell+2}}$ . To enable both  $\hat{b}_{i_\ell}$  and  $\hat{b}_{i_{\ell+1}}$  overlap with  $\hat{a}_{i_{\ell+1}}$ ,  $\hat{a}_{i_{\ell+1}}$  must be in the middle of  $\hat{a}_{i_\ell}$  and  $\hat{a}_{i_{\ell+2}}$  for  $1 \leq \ell \leq z - 1$ . Consequently, for  $1 \leq \ell \leq z - 1$ ,  $\hat{a}_{i_\ell}$  is in the middle of  $\hat{a}_{i_1}$  and  $\hat{a}_{i_{z+1}} = \hat{a}_{i_1}$ , which is impossible.

Statement 2 For any diameter  $S$  of  $G$ , we show that every subtree  $\tau$  hanged on  $S$  must be a dangler. First,  $\tau$  must be hanged on  $S$  at a node in  $A \cup B$ . Otherwise, if  $\tau$  is hanged on  $S$  at a node  $c \in C$ ,  $c$  has degree greater than 2, contradicting Lemma 1. Then,  $\tau$  has more than one node because the root of  $\tau$  is a node in  $C$  and must be of degree 2. If  $\tau$  cannot have more than 2 nodes, Statement 2 follows.



**Fig. 3.** In this example, all  $a_i \in A$ ,  $b_j \in B$  and  $c_k \in C$ .

To prove by contradiction, suppose that  $\tau$  has more than two nodes. Without loss of generality, assume that  $\tau$  is hanged on  $S$  at a node  $a_{i_0} \in A$  and the root of  $\tau$  is a node  $c_{k_3} \in C$ . Note that  $c_{k_3}$  has another neighbour, say  $b_{j_3}$ , from  $B$ . If  $\tau$  contains more than two nodes,  $b_{j_3}$  must have a child, say  $c_{k_6}$ , from  $C$  and  $c_{k_6}$  must have a child, say  $a_{i_3}$ , from  $A$ . Thus,  $\tau$  must have a root-to-leaf path of length more than 4. Then, the two paths from  $a_{i_0}$  to both ends of  $S$  must be of length more than 4. Otherwise,  $S$  cannot be a diameter of  $G$ . From those observations,  $G$  has the pattern shown in Figure 3. According to the pattern,  $\hat{b}_{j_1}, \hat{b}_{j_2}$  and  $\hat{b}_{j_3}$  overlap with  $\hat{a}_{i_0}$ . Therefore, in any valid permutation, one of

$\widehat{b}_{j_1}, \widehat{b}_{j_2}$  and  $\widehat{b}_{j_3}$ , say  $\widehat{b}_{j_2}$ , must be in the middle of the other two fragments and  $\widehat{b}_{j_2}$  can only overlap with  $\widehat{a}_{i_0}$ . However, according to the pattern in Figure 3, for  $\ell = 1, 2, 3$ ,  $\widehat{b}_{j_\ell}$  overlaps with another fragment  $\widehat{a}_{i_\ell}$ , reaching a contradiction.

Now, we know that if  $\mathcal{P}$  has a valid permutation,  $G$  satisfies the two properties of Lemma 2. The remainder of this section show that the converse of this statement is also true. Suppose that  $G$  is a spanning tree with a diameter  $S$  such that all the subtrees hanged on  $S$  are danglers. We define  $\pi_C$  to be a permutation on  $C$  formed by a search defined below.

*Dangler-first search:* Traverse  $G$  starting from one end of  $S$  to the other end of  $S$ ; read off the nodes in  $C$  on  $S$ ; whenever meet any node  $x$  with degree greater than 2, read off the nodes in  $C$  in the danglers hanged on  $S$  at  $x$  in any order and continue to traverse  $S$ .

**Lemma 3.** *The elements in each  $AB_i$  form a consecutive subsequence in  $\pi_C$ . Similarly, the elements in each  $BA_j$  form a consecutive subsequence in  $\pi_C$ .*

*Proof.* For each  $i$ , if  $AB_i$  contains only one element, then the lemma follows. Otherwise,  $a_i$  is of degree at least 2. Then,  $a_i$  must be on the diameter  $S$ . Let  $c$  and  $c'$  be elements in  $AB_i$  which are the two neighbours of  $a_i$  on  $S$ . The remaining nodes in  $AB_i$  must be located in the danglers hanged on  $S$  at  $a_i$ . By dangler-first search, all the elements in  $AB_i$  must form a consecutive subsequence in  $\pi_C$ . By symmetry, for each  $j$ , the elements in  $BA_j$  must form a consecutive subsequence in  $\pi_C$ .

By Lemma 3,  $\pi_C$  can be partitioned into  $p$  subintervals such that the  $r$ th interval contains the elements in  $AB_{i_r}$  for  $r = 1, \dots, p$ . Let  $\pi_A$  be the permutation  $(a_{i_1}, \dots, a_{i_p})$ . Similarly,  $\pi_C$  can be partitioned into  $q$  intervals such that the  $s$ th interval contains the elements in  $BA_{j_s}$  for  $s = 1, \dots, q$ . Let  $\pi_B$  be the permutation  $(b_{j_1}, \dots, b_{j_q})$ . We call  $(\pi_A, \pi_B)$  the *induced permutation* of  $\pi_C$ .

**Lemma 4.** *The induced permutation  $(\pi_A, \pi_B)$  of  $\pi_C$  is a valid permutation of  $\mathcal{P}$ .*

*Proof.* Suppose the lengths from  $A, B$  and  $C$  are plotted on the same line according to the order given by  $\pi_A$ ,  $\pi_B$  and  $\pi_C$ , respectively. Consider the stripes formed from  $A$  and  $C$ . By Fact 1 and Lemma 3, for each  $i$ ,  $\widehat{a}_i$  overlaps with  $\widehat{c}$  for all  $c \in AB_i$ . By symmetry, for each  $j$ ,  $\widehat{b}_j$  overlaps with  $\widehat{c}$  for all  $c \in BA_j$ . Then, by the definition of the EDD problem,  $(\pi_A, \pi_B)$  is a valid permutation.

**Theorem 1.** *Given the enhanced double digest problem  $\mathcal{P}$  and its corresponding graph  $G$ ,  $\mathcal{P}$  has a valid permutation if and only if  $G$  satisfies the two properties in Lemma 2.*

*Proof.* The only-if part follows from Lemma 2. The if part follows from Lemma 4.

### 3.2 A Linear-Time Algorithm for a Duplicate-Free $C$

This section describes how to compute a valid permutation of  $\mathcal{P}$  in  $O(n)$  time. The algorithm is as follows.

**Algorithm** Enhanced-Double-Digest

1. Construct the graph  $G$  corresponding to  $\mathcal{P}$ .
2. If  $G$  does not satisfy the two properties in Lemma 2, then return “no valid permutation”.
3. Find the permutation  $\pi_C$  using dangler-first search.
4. Find the induced permutation  $(\pi_A, \pi_B)$  of  $\pi_C$ .
5. Return  $(\pi_A, \pi_B)$ .

**Lemma 5.** *Algorithm Enhanced-Double-Digest can correctly find a valid permutation in  $O(n)$  time.*

*Proof.* First, by Lemma 4 and Theorem 1, Enhanced-Double-Digest is correct. As for its time complexity, Step 1 requires  $O(n)$  time as  $G$  contains  $2n$  edges and we can find each edge in  $O(1)$  time. Step 2 checks whether  $G$  satisfies the two properties in Lemma 2. For property 1, we can determine whether a graph is a spanning tree in  $O(n)$  time. For property 2 we can compute a diameter of a tree in linear time first, then, we verify whether  $G$  satisfies property 2 by detecting whether the subtrees hanged on the diameter are danglers. Thus, Step 2 requires  $O(n)$  time. Step 3 finds  $\pi_C$  using dangler-first search. Since the search scans every node in  $G$  once, it runs in  $O(n)$  time. Step 4 finds the induced permutation  $(\pi_A, \pi_B)$  of  $\pi_C$  in  $O(n)$  time. In summary, a valid permutation of  $\mathcal{P}$  can be computed in  $O(n)$  time.

To get all valid permutations of  $\mathcal{P}$ , we can modify the dangler-first search to return all possible permutations  $\pi_C$  in a straightforward manner. The induced permutations  $(\pi_A, \pi_B)$  of all such  $\pi_C$  are all valid permutations of  $\mathcal{P}$ .

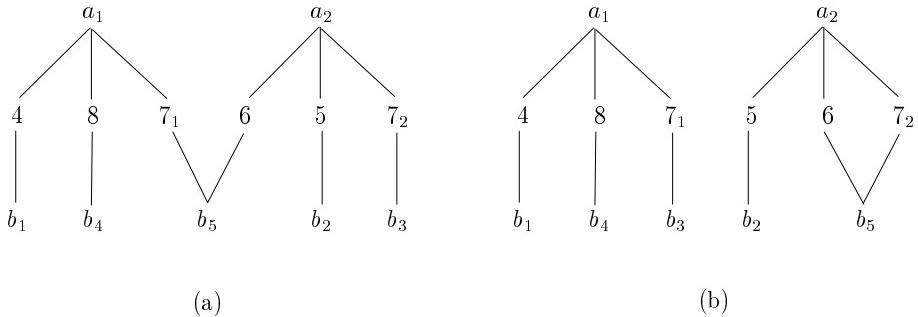
### 3.3 A General Algorithm for $C$ with Few Duplicates

The algorithm Enhanced-Double-Digest in Section 3.2 can solve the EDD problem if  $C$  contains no duplicates. Here, we give an algorithm which works without this assumption.

First, we consider the following example.

- $A = \{a_1 = 18, a_2 = 19\}; B = \{b_1 = 4, b_2 = 5, b_3 = 7, b_4 = 8, b_5 = 13\};$
- $AB_1 = \{5, 6, 7\}; AB_2 = \{4, 7, 8\};$
- $BA_1 = \{4\}; BA_2 = \{5\}; BA_3 = \{7\}; BA_4 = \{8\}; BA_5 = \{6, 7\}.$

In this example, there are two 7's in  $C = \cup_i AB_i = \cup_j BA_j$ . These two 7's in fact represent two different subfragments in the target DNA sequence. To distinguish them, let the copy of 7 in  $AB_1$  be  $7_1$  and that in  $AB_2$  be  $7_2$ . Since 7 also belongs to  $BA_3$  and  $BA_5$ , there are two possible combinations, namely,



**Fig. 4.** (a) is the case where  $7_1 \in BA_5$  and  $7_2 \in BA_3$ ; (b) is the case where  $7_1 \in BA_3$  and  $7_2 \in BA_5$ .

(a)  $7_1 \in BA_5$  and  $7_2 \in BA_3$  and (b)  $7_1 \in BA_3$  and  $7_2 \in BA_5$ . Figure 4(a) and 4(b) illustrate the graph  $G$  for both cases; from these two graphs  $G$ , we can obtain a valid permutation from combination (a). Therefore, we can handle duplicates in  $C$  by giving them different subscripts. Then, all the elements in  $C$  are different and we can solve the enhanced double digest problem using the algorithm Enhanced-Double-Digest in Section B.2. More precisely, we have the following algorithm.

1. If  $C$  contains duplicates, then we assign a unique subscript to each duplicate.
  2. For each possible combinations of the subscripts in the duplicates, we execute Enhanced-Double-Digest to compute a valid permutation.

Let  $\ell$  be the number of duplicates in  $C$ . The above algorithm execute Enhanced-Double-Digest for at most  $\ell!$  time. Therefore, a valid permutation can be computed in  $O(\ell!n)$  time. Thus, if  $\ell$  is constant, the generalized algorithm still runs in linear time.

#### 4 The Enhanced Double Digest Problem Is NP-hard

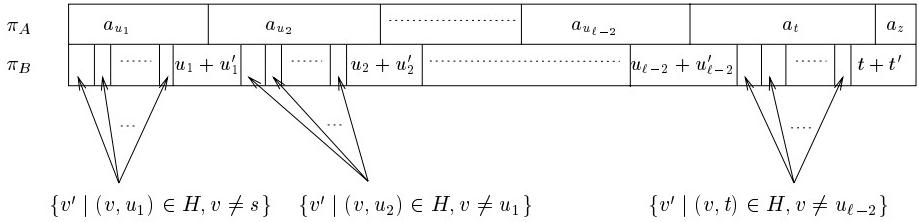
This section proves the NP-hardness of the enhanced double digest problem by a reduction from the Hamiltonian Path problem [2].

Given an undirected graph  $H$ , we show that in polynomial time, we can construct an EDD instance  $\mathcal{Q}$  so that  $H$  contains a hamiltonian path if and only if  $\mathcal{Q}$  has a valid permutation. For ease of prove, we augment  $H$  with two new nodes  $t$  and  $z$ . All nodes originally in  $H$  have edges to  $t$ . In addition, we add an edge  $(t, z)$  to  $H$ . Note that the original  $H$  contains a hamiltonian path if and only if the amended  $H$  has a hamiltonian path. Let  $\ell$  be the number of nodes in  $H$ . Assume that the nodes in  $H$  are labeled by  $\{1, 2, \dots, \ell\}$ . For each node  $v$ , let  $\kappa(v)$  be the number of neighbours of  $v$ . Let  $v' = v + \ell$ . The EDD instance  $\mathcal{Q}$  is given the following length information. Note that this length information can be constructed from  $H$  in polynomial time.

- $A = \{a_v \mid v \in H\}$  where  $a_z = t'$ ,  $a_t = t + \sum_{u \in H - \{t, z\}} u'$  and  $a_v = v + \sum_{(u, v) \in H} u'$  for  $v \neq z, t$ . Also,  $AB_z = \{t'\}$ ;  $AB_t = \{u' \mid u \in H - \{t, z\}\} \cup \{t\}$ ; and  $AB_v = \{u' \mid (u, v) \in H\} \cup \{v\}$  for  $v \neq z$ .
- $B = \{b_v, b_{v(1)}, \dots, b_{v(k(v)-1)} \mid v \in H - \{z\}\}$  where  $b_v = v + v'$  and  $b_{v(i)} = v'$  for all  $v \in H - \{z\}$  and all  $i \leq \kappa(v) - 1$ . Also,  $BA_v = \{v, v'\}$  and  $BA_{v(i)} = \{v'\}$ .

**Lemma 6.**  $H$  has a hamiltonian path if and only if there is a valid permutation for  $\mathcal{Q}$ .

*Proof.* The two directions are proved as follows.



**Fig. 5.** The permutations  $\pi_A$  and  $\pi_B$  of  $A$  and  $B$ , respectively.

( $\Rightarrow$ ) Let  $u_1, u_2, \dots, u_{\ell-2}, t, z$  be a hamiltonian path in  $H$ . Let  $\pi_A$  and  $\pi_B$  be permutations of  $A$  and  $B$  as shown in Figure 5. It is easy to check that  $(\pi_A, \pi_B)$  is a valid permutation to  $\mathcal{Q}$ .

( $\Leftarrow$ ) Let  $(\pi_A, \pi_B)$  be a valid permutation of  $\mathcal{Q}$ . The remainder of this proof shows that the ordering of the lengths in  $\pi_A$  defines a hamiltonian path in  $H$ .

Assume the lengths from  $A$  are plotted on a line according to the order given by  $\pi_A$  and similarly, the lengths from  $B$  are also plotted on this line according to  $\pi_B$ . For each  $v \in H$ , the line fragment corresponds to  $a_v \in A$  is called  $\hat{a}_v$ . For each  $v \in H - \{z\}$ , the line fragment corresponds to  $b_v \in B$ , is called  $\hat{b}_v$ .

For every  $v \in H - \{z\}$ , since  $BA_v = \{v, v'\}$ ,  $\hat{b}_v$  overlaps with two consecutive line fragments from  $A$ ; in addition, the overlapping regions between  $\hat{b}_v$  and these two line fragments must be of length  $v$  and  $v'$ , respectively. Observe that  $v \in AB_v$  and  $v \notin AB_u$  for all  $u \neq v$ . One of these two fragments, which overlaps with  $\hat{b}_v$ , must be  $\hat{a}_v$ . The other line fragment can be  $\hat{a}_u$  for any  $u \in H$  with  $v' \in AB_u$ , i.e.,  $(v, u) \in H$ .

Let  $\pi_A = (a_{u_1}, \dots, a_{u_\ell})$ . From the above argument, we know that, for every two consecutive line fragments  $\hat{a}_i$  and  $\hat{a}_{i+1}$ , there exists a fragment  $\hat{b}_v$  (where  $v$  is either  $u_i$  or  $u_{i+1}$ ) which overlaps with both  $\hat{a}_{u_i}$  and  $\hat{a}_{u_{i+1}}$ . The above argument also implies that  $(u_i, u_{i+1}) \in H$ . Thus,  $u_1, \dots, u_\ell$  forms a path in  $H$ . As  $u_1, \dots, u_\ell$  contains all the  $\ell$  nodes of  $H$ , this path is a hamiltonian path.

## 5 Further Research Directions

This work can be extended in several directions. One direction is to design a series of laboratory procedures that can actually produce the input length information in the required form. While separating out DNA sequences by length seems to be possible with current laboratory techniques, we still face the problem of separating different DNA fragments having the same length. Another direction is to consider the problem of more than 2 digesting enzymes. Using multiple enzymes could help resolve the issue of multiple solutions that arise when there are dangles or duplicate subfragment lengths. Also, the extra input may actually make the problem solvable in a shorter period of time. The third direction is to have a probabilistic analysis of the number of duplicates in  $C$ , when the length of the target DNA sequence is given. Lastly, this paper does not address the issue of noise in the length data. From the practical point of view, handling the noise problem is quite important.

### Acknowledgments

We thank the anonymous referees for helpful suggestions.

### References

1. W. M. Fitch, T. F. Smith, and W. W. Ralph. Mapping the order of DNA restriction fragments. *Gene*, 22:19–29, 1983.
2. M. Garey and D. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.
3. L. Goldstein and M. S. Waterman. Mapping DNA by stochastic relaxation. *Advances in Applied Mathematics*, 8:194–207, 1987.
4. Richard M. Karp. Mapping the genome: Some combinatorial problems arising in molecular biology. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 278–285, San Diego, California, 16–18 May 1993.
5. D. Nathans and H. O. Smith. Restriction endonucleases in the analysis and restructuring of DNA molecules. *Annual Review of Biochemistry*, 44:273–293, 1975.
6. P. A. Pevzner. DNA physical mapping, flows in networks and minimum cycles mean in graphs. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 8:99–112, 1992.
7. P. A. Pevzner. DNA physical mapping and alternating Eulerian cycles in colored graphs. *Algorithmica*, 13:77–105, 1995.
8. W. Schmitt and M. S. Waterman. Multiple solutions of DNA restriction mapping problems. *Advances in Applied Mathematics*, 12:412–427, 1991.
9. M. Stefk. Inferring DNA structure from segmentation data. *Artificial Intelligence*, 11:85–114, 1978.
10. M. S. Waterman and J. R. Griggs. Interval graphs and maps of DNA. *Bulletin of Mathematical Biology*, 48:189–195, 1986.

# Generalization of a Suffix Tree for RNA Structural Pattern Matching

Tetsuo Shibuya

IBM Tokyo Research Laboratory,  
1623-14, Shimotsuruma, Yamato-shi, Kanagawa 242-8502, Japan.  
[tshibuya@trl.ibm.co.jp](mailto:tshibuya@trl.ibm.co.jp)

**Abstract.** In molecular biology, it is said that two biological sequences tend to have similar properties if they have similar 3-D structures. Hence, it is very important to find not only similar sequences in the string sense, but also structurally similar sequences from databases. In this paper, we propose a new data structure that is a generalization of a parameterized suffix tree (p-suffix tree for short) introduced by Baker. This data structure can be used for finding structurally related patterns of RNA or single-stranded DNA. Furthermore, we propose an  $O(n(\log |\Sigma| + \log |\Pi|))$  on-line algorithm for constructing it, where  $n$  is the sequence length,  $|\Sigma|$  is the size of the normal alphabet, and  $|\Pi|$  is that of the alphabet called “parameter,” which is related to the structure of the sequence. Our algorithm achieves a linear time when it is used to analyze RNA and DNA sequences. Furthermore, as an algorithm for constructing the p-suffix tree, it is the first on-line algorithm, though the computing bound of our algorithm is same as that of Kosaraju’s best-known algorithm. The results of computational experiments using actual RNA and DNA sequences are also given to demonstrate our algorithm’s practicality.

## 1 Introduction

The 3-D structure of a biological sequence plays a major role in determining its functions and properties, and sequences that have similar structures often have similar functions, even if the sequences themselves are not similar. But it is very difficult to predict the structure of a given sequence correctly and efficiently. Hence it seems to be still harder to find structurally similar regions among several biological sequences or to find a set of frequently appearing and structurally similar regions in a given sequence. Thus molecular biologists often search for only similar, or highly conserved regions from DNA, RNA or protein sequences to find regions with similar functions, because similar sequences have tendency of having the same structure. Though many such methods are very fast, they do not detect regions that are structurally similar to each other but not similar in the string sense.

RNA sequences consist of four kinds of bases: A (adenine), U (uracil), C (cytosine), and G (guanine). Note that in DNA, T (thymine) is present instead of U. A and U (T for DNA) are said to be complements of each other, and C and G are

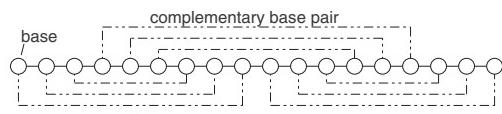
also complementary bases. RNA and single-stranded DNA sequences often form some structures by combining two complementary base pairs. It is known that double-stranded DNA sequences sometimes form such structures by becoming single-stranded locally. Note that a base sometimes combines with more than one complementary base: The triplex structure is the famous example. Many computational studies have been done to predict RNA secondary structure, comparing a new sequence with a known RNA structure, searching a known RNA or DNA structures from large databases, and so on [2][10][12][13][16][17][19][20]. But there has been no appropriate method that can mine an unknown important RNA structure from a large data set efficiently in a linear time, which is the aim of the algorithm presented in this paper.

Let us consider the two RNA sequences in Figure 1(1). The two sequences are not at all similar to each other: there are no identical bases in identical positions. In sequence 1, A's are located at the 1st, 3rd, 8th, and 15th positions. In sequence 2, C's are located at the same position as A's in sequence 1. Similarly, A's, U's, and G's in sequence 2 are located at the same positions as G's, C's, and U's in sequence 1, respectively. Recall that A and U can combine with each other, and that C and G can also combine with each other. We then notice the following fact: If two bases in one of these sequences can combine with each other, then in the other sequence, two bases at in same two positions are also able to combine with each other. This implies that a structure that can be formed by one of the sequences can also be formed by the other sequence. Thus there is a strong possibility that these two sequences have the same structure, and consequently may have similar properties. For example, Figure 1(2) shows one of the structures that can be formed by sequence 1. It is easy to see that it can also be formed by sequence 2.

In this paper, we first introduce suffix trees and p-suffix trees as preliminaries. We also briefly describe Ukkonen's algorithm, on which our algorithm is based. We then propose a new data structure called an s-suffix tree by generalizing the p-suffix tree. We also discuss how to describe structural patterns of RNA or DNA here. Using the s-suffix tree, we can efficiently find some set of substrings in some given sequence that might be structurally similar, query substrings that might be structurally similar to another given string, and so on. We also propose an efficient on-line algorithm for constructing an s-suffix tree based on Ukkonen's algorithm. Finally, we give the results of computational experiments using several HIV RNA complete sequences and very large DNA sequences of *E. coli* (*Escherichia coli*).

Sequence 1: AUAUCGUUAUGGCCGAGCC  
Sequence 2: CGCGUAGCGAAUUACAUU

(1) Example sequences



(2) Candidate structure

**Fig. 1.** Examples of sequences that have high possibility to have a same structure

## 2 Preliminaries

### 2.1 Suffix Trees and p-Suffix Trees

The suffix tree of a string  $S \in \Sigma^n$  is the compacted trie of all the suffixes of  $S\$$  ( $\$ \notin \Sigma$ ) [9,10,15,18,21]. This data structure is very useful for various problems in sequence pattern matching. Using it, we can query a substring of length  $m$  in  $O(m \log |\Sigma|)$  time, we can find frequently appearing substrings in a given sequence in linear time, we can find a common substring of many sequences, also in linear time, and so on [10].

The tree has  $n + 1$  leaves, and each internal node has more than one child. Each edge is labeled with a non-empty substring of  $S\$$ , and no two edges out of a node can have labels that start with the same character. Each node is labeled with the concatenated string of edge labels on the path from the root to the node, and each leaf has a label that is a different suffix of  $S\$$ . Because each edge label is represented by the first and the last indices of the corresponding substring in  $S\$$ , the data structure can be stored in  $O(n)$  space.

This data structure was first proposed by Weiner [21], who gave an  $O(n|\Sigma|)$  algorithm for constructing it, where  $n$  is the string length and  $|\Sigma|$  is the size of the alphabet. McCreight [15] improved it by giving an  $O(n \log |\Sigma|)$  algorithm. After that, Ukkonen [18] proposed an on-line  $O(n \log |\Sigma|)$  algorithm, which processes a string character by character from left to right. Recently, Farach [9] proposed an  $O(n)$  algorithm for an integer alphabet  $\{1, \dots, n\}$ .

A parameterized string, or a p-string for short, is a string over two alphabets  $\Sigma$  and  $\Pi$ , where  $\Sigma$  is an ordinary alphabet and  $\Pi$  is a set of parameters. Two p-strings are said to match if they are same except for a one-to-one correspondence between the characters in  $\Pi$  occurring in them. For example, two p-strings  $ACxBCzyAzxC$  and  $ACyBCzzAxzC$  match ( $\Sigma = \{A, B, C\}$  and  $\Pi = \{x, y, z\}$ ).

As in [5], we define  $\text{prev}(S)$  for any p-string  $S$  as follows:

**Definition 1.** Let  $N$  be the set of nonnegative integers. For any parameter  $x \in \Pi$  in string  $S \in (\Sigma \cup \Pi)^*$ , replace it by an integer in  $N$  that equals the number of positions between it and the nearest  $x$  to the left, except for the leftmost  $x$ , which is replaced by 0. We let the obtained string in  $(\Sigma \cup N)^*$  be  $\text{prev}(S)$ .

For example,  $\text{prev}(ACxBCzyAzxC) = AC0BC002A38C$ . The p-suffix tree of a p-string  $S$  is the compacted trie for all  $\text{prev}(\text{suffix}_i(S))$  for all positions  $i$ , where  $\text{suffix}_i(S)$  denotes a suffix of  $S$  that starts at position  $i$ . Baker [3,5,6] proposed this data structure and showed that it can be constructed in  $O(n(|\Pi| + \log |\Sigma|))$  time. Kosaraju [14] improved the time by giving an  $O(n(\log |\Pi| + \log |\Sigma|))$  algorithm. Note that both of the algorithms are based on McCreight's suffix tree construction algorithm [15] and that neither supports on-line computation. This paper will give an on-line algorithm for the same task, based on Ukkonen's algorithm [18].

In the following sections, we use the following definitions. In a suffix tree, let  $\text{parent}(u)$  be the parent node of node  $u$ , let  $\sigma_u$  be the string label of node  $u$ , and let  $\text{node}(\alpha)$  be node  $u$  in the tree such that  $\sigma_u = \alpha$  if it exists. The suffix link

of  $u$  is a link to a node with label  $\alpha$  if  $u$  is not the root, and has a label of  $c\alpha$ , where  $c$  is a single character. It is known that a suffix link always exists for any  $u$  except for the root in a suffix tree [10,15,18]. If  $u$  is the root we let its suffix link be  $u$  itself. Let  $sl(u)$  be the suffix link of  $u$ .

## 2.2 Ukkonen's Suffix Tree Construction Algorithm

In this section, we briefly describe Ukkonen's suffix tree construction algorithm.

The implicit suffix tree of  $S$  is the compacted trie of all the suffixes of  $S$ , and a label for an edge that ends at a leaf is represented by only the first index of the label. Let  $\text{prefix}_i(S)$  be a prefix of  $S$  whose length is  $i$ , let  $T_i$  denote the implicit suffix tree of  $\text{prefix}_i(S\$)$ , and let  $n = |S|$ . Ukkonen's algorithm consists of  $n + 1$  phases, and in the  $i$ th phase, we construct an implicit suffix tree  $T_i$  from  $T_{i-1}$ .

In the  $i$ th phase, we construct a new node  $u = \text{node}(\text{suffix}_j(\text{prefix}_i(S)))$  for all  $1 \leq j \leq i$  in this order if there is no locus for  $\text{suffix}_j(\text{prefix}_i(S))$  in the tree. When we construct  $u$ , if there is no node with a label of  $\text{suffix}_{j-1}(\text{prefix}_i(S))$ , we must also construct a new internal node at the appropriate locus and let it be the parent of  $u$ . We call this procedure for single  $j$  the  $j$ th extension of the  $i$ th phase.

Notice that we do not have to construct node  $u = \text{node}(\text{suffix}_j(\text{prefix}_i(S)))$  if  $v = \text{node}(\text{suffix}_j(\text{prefix}_{i-1}(S)))$  was a leaf in the previous phase, because of the definition of the implicit suffix tree:  $\sigma_v$  is  $\text{suffix}_j(\text{prefix}_i(S))$  in this phase. Thus, if there is a leaf for each of  $\text{node}(\text{suffix}_j(\text{prefix}_{i-1}(S)))$  for all  $j < k$  in phase  $i - 1$ , we can begin by constructing  $\text{node}(\text{suffix}_{j+1}(\text{prefix}_i(S)))$  in this phase. Furthermore, if there is a locus for  $\text{suffix}_j(\text{prefix}_i(S))$  for some  $j$ , it is easy to see that there already exist loci for  $\text{suffix}_k(\text{prefix}_i(S))$  ( $k > j$ ) too, and that there is no need to construct nodes for them in this phase.

Ukkonen's algorithm, like McCreight's algorithm, maintains at each node  $u$  of the suffix tree a suffix link  $sl(u)$ . In any phase, we construct nodes  $u_j = \text{node}(\text{suffix}_j(\text{prefix}_i(S)))$  for several consecutive  $j$ 's and  $u'_j = \text{node}(\text{suffix}_j(\text{prefix}_{i-1}(S)))$  if necessary, in the manner described above. Notice that  $u_{j+1} = sl(u_j)$  and  $u'_{j+1} = sl(u'_j)$  if they exists. For the last  $u_j$  to be constructed in this phase, we will check the locus for  $\text{suffix}_{j+1}(\text{prefix}_i(S))$ , which is  $sl(u_j)$  in the next extension according to the algorithm. Thus we will know within the phase the suffix links of all the constructed nodes in the same phase. In this way, we can maintain the suffix links.

Using the suffix links, we can construct node  $u_j = \text{node}(\text{suffix}_j(\text{prefix}_i(S)))$  faster: It is easy to see that  $sl(\text{parent}(u_{j-1}))$  must be an ancestor of  $u_j$ , and we can find the locus of  $\text{suffix}_j(\text{prefix}_{i-1}(S))$  by tracing edges from  $sl(\text{parent}(u_{j-1}))$ . We call tracing from the suffix link to the target locus "scanning."

In this way, the algorithm achieves an  $O(n \log |\Sigma|)$  time complexity. For more details of the algorithm and the analysis of the computing time bound, see [10] or [18].

### 3 Structural Suffix Tree

#### 3.1 s-Strings and s-Suffix Trees

In this section we define s-strings and s-suffix trees, which are generalizations of p-strings and p-suffix trees.

**Definition 2.** Let  $\Sigma$  and  $\Pi$  be disjoint finite alphabets. We call the characters in  $\Sigma$  the “fixed symbols” and those in  $\Pi$  the “parameters.” Some of the characters in  $\Pi$  have one-to-one correspondences to other characters in  $\Pi$ : No two characters can be complements of one character, and two characters that correspond to each other are called complementary characters. A string in  $(\Sigma \cup \Pi)^*$  is called a structural string, or s-string for short. Two s-strings  $S$  and  $S'$  are said to s-match if they satisfy the following two conditions: (1) there exists a one-to-one mapping from  $\Pi$  to  $\Pi$  such that  $S$  becomes  $S'$  as a result of applying it, and (2) if  $x$  is mapped to  $y$  in the mapping, then the complement of  $x$  is also mapped to the complement of  $y$  in the mapping.

For example, if  $\Sigma = \{A, B\}$ ,  $\Pi = \{x, y, z, w\}$ , and  $x$  and  $y$  are complements of  $z$  and  $w$ , respectively, then  $ABxByAzwz$  and  $ABwBxAyzy$  s-match, but  $ABxByAzwz$  and  $ABwBxAxyz$  do not. Note that if there are no complementary pairs in  $\Pi$ , an s-string is the same as a p-string. Note also that the complement of a given character can be accessed in  $O(\log |\Pi|)$  time if the information is stored in a balanced tree data structure, which can be constructed in  $O(|\Pi| \log |\Pi|)$  time. If  $\Pi$  can be used as an index to a table, the complement can be obtained in  $O(1)$  time.

The problem of the RNA (or DNA) structural matching described in section 1 is the problem of s-matching in the following situation:  $\Sigma = \emptyset$ ,  $\Pi = \{A, U, C, G\}$ , and  $A$  and  $C$  are complementary characters of  $U$  and  $G$ , respectively. If two RNA sequences s-match with each other, it can be said that there is a high possibility that the two sequences have the same structure and that they may have similar properties as a result. For example, the two sequences in Figure 1 (1) s-match.

The following two encodings are useful for determining s-matching of two sequences. One is  $\text{prev}(S)$  that is already defined in Definition 1. The other is  $\text{compl}(S)$  defined as follows:

**Definition 3.** Let  $N$  be the set of nonnegative integers ( $N \notin \Sigma \cup \Pi$ ). For any parameter  $x \in \Pi$  in string  $S \in (\Sigma \cup \Pi)^*$ , replace it by an integer in  $N$  that equals the number of positions between it and the nearest complementary character of  $x$  to the left. If there is no complementary character to the left, replace it by 0. Let  $\text{compl}(S)$  denote the obtained string in  $(\Sigma \cup N)^*$ .

For example,  $\text{compl}(ABxByAzwz) = AB0B0A436$  if  $\Sigma = \{A, B\}$ ,  $\Pi = \{x, y, z, w\}$ , and  $x$  and  $y$  are complements of  $z$  and  $w$ , respectively. We can compute  $\text{prev}$  and  $\text{compl}$  encodings for string  $S$  of size  $n$  in  $O(n \cdot \min(\log n, \log |\Sigma|))$  time and  $O(n)$  space by means of a balanced tree structure, which can be computed on-line. If

$\Pi$  is known and can be used as an index to a table of size  $|\Pi|$ , it is easy to see that these encodings can be computed in  $O(n + |\Pi|)$  time and space.

These two encodings are related to finding s-matches as follows: s-strings  $S$  and  $S'$  are an s-match if and only if  $\text{prev}(S) = \text{prev}(S')$  and  $\text{compl}(S) = \text{compl}(S')$ . Furthermore, it is easy to see the following lemma. Let  $\text{prev}(S)[i]$  and  $\text{compl}(S)[i]$  denote the  $i$ th characters of  $\text{prev}(S)$  and  $\text{compl}(S)$  respectively.

**Lemma 1.** *Consider a situation in which  $\text{prefix}_i(S)$  and  $\text{prefix}_i(S')$  are an s-match. In this situation, if  $\text{prev}(S)[i+1] = \text{prev}(S')[i+1] \neq 0$ ,  $\text{compl}(S)[i+1] = \text{compl}(S')[i+1]$ . Similarly, if  $\text{compl}(S)[i+1] = \text{compl}(S')[i+1] \neq 0$ ,  $\text{prev}(S)[i+1] = \text{prev}(S')[i+1]$ .*

This means that, when we check s-matches of strings, we do not have to see the other encoding if one of the encodings encodes a character as a non-zero number. Using this lemma, we can check s-matching by using the following s-encoding:

**Definition 4.** *For a given string  $S$ , compute  $\text{prev}(S)$  and  $\text{compl}(S)$ . If  $\text{prev}(S)[i] = 0$ , replace it with  $-\text{compl}(S)[i]$ , which is a nonpositive value. We call this new encoded string in  $(\Sigma \cup I)^*$  ( $I$ : integer) as a structural encoding of  $S$ , or an s-encoding for short.*

The structural suffix tree of string  $S$ , or the s-suffix tree of  $S$  for short, is the compacted trie of the s-encoded strings of all the suffixes of  $S$ . Let  $\text{sencode}(S)$  denote the s-encoding of  $S$ . The s-strings  $S$  and  $S'$  are an s-match if and only if  $\text{sencode}(S) = \text{sencode}(S')$ . Let  $\text{ssuffix}_i(S) = \text{sencode}(\text{suffix}_i(S))$ , and let  $\text{ssuffix}_i(S)[j]$  be the  $j$ th character of  $\text{ssuffix}_i(S)$ . Notice that  $\text{ssuffix}_i(S)[j]$  is sometimes different from  $\text{sencode}(S)[i+j-1]$ : if  $|\text{sencode}(S)[i+j-1]| > j$ ,  $\text{ssuffix}_i(S)[j] = 0 \neq |\text{sencode}(S)[i+j-1]| > j$ . Notice also that, if we have  $\text{prev}(S)$  and  $\text{compl}(S)$ , we can obtain the value of  $\text{ssuffix}_i(S)[j]$  for any  $i$  and  $j$  in a constant time.

Using the s-suffix tree, we can find some set of substrings in some given sequence that s-match each other in  $O(n)$  time or query a substring that matches another given string in  $O(n \log(|\Sigma| + |\Pi|))$  time. We can also find common s-substrings of given two sequences in a linear time.

### 3.2 Basic Algorithm

We first describe a basic method for constructing the s-suffix tree based on Ukkonen's algorithm.

The implicit s-suffix tree of  $S$  is the compacted trie of all the s-encoded suffixes of  $S$ , and a label for an edge that ends at a leaf is represented by only the first index of the label in it. Let  $T_i$  denote the implicit s-suffix tree of  $\text{prefix}_i(S\$)$  for the given string  $S$  and an integer  $0 < i \leq n+1$  where  $n = |S|$ . Let  $\text{node}(S)$  denote the node with label of s-encoded string of  $S$  in this section. Like Ukkonen's algorithm, our basic algorithm consists of  $n+1$  phases, and in the  $i$ th phase, we construct an implicit s-suffix tree  $T_i$  from  $T_{i-1}$ .

As in Ukkonen's algorithm, we construct a new node  $u = \text{node}(\text{suffix}_j(\text{prefix}_i(S)))$  for all  $1 \leq j \leq i$  in this order if there is no locus for  $\text{suffix}_j(\text{prefix}_i(S))$  in the tree in the  $i$ th phase. We call this procedure for single  $j$  the  $j$ th extension as in the description of Ukkonen's algorithm. Ukkonen's algorithm speeds up each phase by ignoring unnecessary extensions. In this s-suffix tree case, the unnecessary extensions are all the same as Ukkonen's case.

The major problem in constructing s-suffix trees by applying Ukkonen's algorithm is that, as in Baker's p-suffix tree construction algorithm, a node of an s-suffix tree does not always have explicit suffix links to another node. Consider a node  $u = \text{node}(c\alpha)$  in an s-suffix tree, where  $c$  is a single character and  $\alpha$  is some s-encoded s-string. It is possible that the locus for  $\alpha$  is not a node but a point on an edge. In this case, we let  $u$ 's suffix link  $sl(u)$  be this edge and call such a link an implicit suffix link.

The implicit suffix links causes two problems. One is how to keep these implicit suffix links correct thorough the algorithm: the implicit suffix links must be updated if the corresponding edge is split. The other is how to analyze the number of scanned nodes in the algorithm. First, we deal with the former problem, and after that we discuss the latter problem.

It is easy to see the following lemma related to implicit suffix links:

**Lemma 2.** *Let  $u$  be a node with an implicit suffix link and  $d = |\sigma_u|$ . Then the first s-encoded character of the label of any of the outgoing edges from  $u$  must be one of  $d$ ,  $0$ , and  $-d$ . Furthermore, if it is  $d$ , its corresponding  $\text{compl}$  value must be  $0$ .*

We use the term ‘zero-node’ for a node with more than one outgoing edge that has a label starting with either of  $d$ ,  $0$ , or  $-d$ , where  $d$  is the label length of the node, regardless of whether its suffix link is implicit or not. We also call edges the first s-encoded character of whose labels are  $d$ ,  $0$  and  $-d$ , a “positive zero-edge”, a “normal zero-edge” and a “negative zero-edge,” respectively.

The following lemmas related to zero-nodes and zero-edges can be easily seen:

**Lemma 3.** *On a path from the root to a leaf, there are at most  $|\Pi|$  zero-nodes.*

**Lemma 4.** *A positive zero-edge cannot be an ancestor of another positive zero-edge. Similarly, a negative zero-edge cannot be an ancestor of another negative zero-edge.*

It is easy to find the implicit suffix links of newly constructed nodes in the algorithm, as in Ukkonen's algorithm. Consider the situation in the  $j$ th extension of the  $i$ th phase of the algorithm, when  $u_j = \text{node}(\text{suffix}_j(\text{prefix}_i(S)))$  is constructed. Note that  $u'_j = \text{node}(\text{suffix}_j(\text{prefix}_{i-1}(S)))$  may be constructed at the same time if necessary. As in the case of constructing an ordinary suffix tree, we will check the locus of  $\text{suffix}_{j+1}(\text{prefix}_{i-1}(S))$  in the next extension in the same phase, so we will soon find the suffix links of  $u_j$  and  $u'_j$ . Hence we can

conclude that every node other than the last leaf inserted and its parent has either an ordinary suffix link or an implicit suffix link. The problem is how to maintain these implicit suffix links. In the algorithm, we often split edges to add a new node. Thus we have to update each implicit link if the edge it links to is split into two edges by inserting a new node. We call a set of nodes a zero-chain if the nodes form a chain in the tree and all the edges between them are the same kind zero-edges (*i.e.*, if one edge is a positive zero-edge, then the others are also positive zero-edges, for example).

We obtain the following theorem related to implicit suffix links:

**Theorem 1.** *For any edge  $e$ , the set of nodes having implicit suffix links to  $e$  forms at most  $2|\Pi| + 1$  zero-chains in the tree. Furthermore, the length of each zero-chain is at most  $|\Pi|$ .*

*Proof.* Let  $v$  and  $v'$  be two nodes with implicit suffix links to the same edge. If  $v$  is an ancestor of  $v'$  and there is a node  $u$  between  $v$  and  $v'$ , it is obvious that  $sl(u)$  is also between  $sl(v)$  and  $sl(v')$ .

If neither of these two nodes is an ancestor of the other, let  $w$  be the lowest common ancestor of  $v$  and  $v'$  in the suffix tree. Note that  $w$  is not the root, because both of the first s-encoded characters of the labels of  $v$  and  $v'$  must be 0. Since one of the s-encoded strings of  $\text{suffix}_2(\sigma_v)$  and  $\text{suffix}_2(\sigma_{v'})$  must be a prefix of the other, the outgoing edges to  $v$  and  $v'$  must be zero-edges. Thus  $w$  must be a zero-node.

Lemma 4 implies that, under the negative or positive zero-edge out of  $w$ , there is only one zero-chain formed by the set of nodes having implicit suffix links to  $e$ . Furthermore, a normal zero-edge can have at most  $|\Pi| - 1$  normal zero-edges on a path to some leaf from it, according to Lemma 3. Thus we can conclude that there are at most  $2|\Pi| + 1$  zero-chains. Also according to Lemma 3, it is obvious that the lengths of the zero-chains are at most  $|\Pi|$ .

Note that, in the case of the p-suffix tree (*i.e.*, when there are no complementary character pairs), such nodes form only one zero-chain. According to this theorem, there are at most  $2|\Pi|^2 + |\Pi|$  implicit suffix links to one edge. Hence when we split an edge, it takes  $O(|\Pi|^2)$  time to update all the corresponding implicit suffix links if we do it naively. If  $|\Pi|$  is constant, the bound is  $O(n)$ , but if  $|\Pi|$  is large, it causes a problem. From now on, we consider how to reduce the time to  $O(\log |\Pi|)$ .

Consider  $O(n)$  sets of nodes which are empty at first. We perform two types of procedures for the sets. One is inserting a node into one of the sets, and the other is splitting one of the sets into two sets according to the label lengths of the nodes in the set, as follows: One of the two sets newly constructed by splitting is the set of nodes whose label lengths are larger than a specified length, and the other is the set of nodes whose label lengths are smaller than the same specified length. Consider that the upper bound of the size of a set is  $p$ , and that the total number of nodes inserted is  $O(n)$ . A balanced data structure can achieve the following time bounds for these two procedures:

1. A new item can be inserted into a set in  $O(\log p)$  time.
2.  $S$  can be split into two sets as above in a time linear to the size of the smaller set of them.

It is easy to see that the total time taken by procedure 1 is  $O(n \log p)$ , and that the total time taken by procedure 2 is also  $O(n \log p)$ . We can maintain the implicit suffix links by this procedure. In this case,  $p = O(|\Pi|^2)$ ; thus the total time required for maintaining them is  $O(n \log |\Pi|)$ .

From now on, we discuss the number of nodes scanned in the algorithm. Note that it is  $O(n)$  in Ukkonen's algorithm [10, 18]. Note also that the analysis for it is almost the same as that of the number of rescanned nodes in Baker's algorithm for p-suffix trees [5, 6].

**Theorem 2.** *The number of scanned edges that are not normal zero-edges is at most  $n$ .*

*Proof.* In constructing the s-suffix tree of a string  $S$ , consider that an edge  $(u, v)$  that is not a normal zero-edge is scanned when we search for the locus of  $\text{suffix}_j(\text{prefix}_i(S))$  in the  $j$ th extension of the  $(i+1)$ th phase. Let  $u = \text{node}(\text{suffix}_j(\text{prefix}_{k-1}(S)))$ .

Consider the locus of  $\text{suffix}_{j'}(\text{prefix}_{k-1}(S))$  for any  $j' < j$ . Note that we do not perform the  $j''$ th ( $j'' \geq j$ ) extension in the  $i'$ th phase ( $i' < i$ ) of the algorithm. If there exists a node ( $w$ ) for the locus, then it must have an explicit suffix link to some node, because  $\text{suffix}_{j'}(\text{prefix}_k(S))[k - j' + 1]$  is not 0. This means that  $w$  cannot be scanned in the algorithm. Accordingly, we conclude that the number of such edges is at most  $n$ .

According to Lemma 3, the number of normal zero-edges that are scanned in a single phase is at most  $|\Pi|$ . Thus the total number of nodes that have implicit suffix links and are scanned in the algorithm is at most  $n|\Pi|$ . Note that the outgoing normal zero-edge from a node can be accessed in  $O(1)$  time. Thus the total scanning time will be  $O(n(|\Pi| + \log |\Sigma|))$ .

Thus we conclude that the total computing time of our algorithm is  $O(n(|\Pi| + \log |\Sigma|))$ . If  $|\Sigma|$  and  $|\Pi|$  are constant, it is  $O(n)$ . In fact, in the problem of RNA/DNA structural matching ( $|\Sigma| = 0$  and  $|\Pi| = 4$ ), this basic algorithm is efficient enough.

### 3.3 Faster Algorithm when $|\Sigma| = 2$

In this section, we will improve the algorithm given in the previous section to  $O(n \log |\Pi|)$  when  $|\Sigma| = 2$ . The technique used in this section is almost the same as Kosaraju's technique [14] for improving of the rescanning procedure of Baker's algorithm.

In each extension, if we insert a new node into an edge  $(u, v)$ , we will scan from  $sl(u)$  to find the locus of the suffix link of the new node. We want to reduce the number of zero-edges encountered in this scanning. Let  $Z_{uv}$  be the set of

normal zero-edges whose starting node is encountered in scanning from  $sl(u)$  to  $sl(v)$ . Note that  $|Z_{uv}| \leq |\Pi|$ .

For each edge, we maintain a concatenable queue [114], or c-queue for short. Each c-queue for  $(u, v)$  contains a set of edges in  $Z_{uv}$  arranged in the order of their depth. We maintain the c-queues lazily; that is, we put edges into c-queues only when we first encounter the edges in scanning. Thus the c-queue for edge  $(u, v)$  does not contain all of the edges in  $Z_{uv}$ . The same edge can appear only in 2 c-queues because  $|\Sigma| = 2$ .

The time taken to insert an edge into a c-queue is  $O(\log |\Pi|)$ . In scanning for the locus of depth  $d$ , we begin from the deepest edge in the corresponding c-queue whose starting depth is shallower than  $d$ . Such an edge can be found in  $O(\log |\Pi|)$  time. Furthermore, we must split the c-queue when the edge is split, and this can be done in  $O(\log |\Pi|)$  time.

In this way, we can achieve an  $O(n \log |\Pi|)$  time algorithm if  $|\Sigma|$  is a small constant. Note that the space complexity is  $O(n)$ . From now on, we consider how to achieve  $O(n(\log |\Pi| + \log |\Sigma|))$  time.

### 3.4 Faster Algorithm for Arbitrary $\Sigma$

In this section, we will improve the algorithm given in the previous sections to  $O(n(\log |\Sigma| + \log |\Pi|))$ , which is far more efficient for larger alphabets. The technique used in this section is also almost the same as Kosaraju's technique [14] for improving the rescanning procedure of Baker's algorithm, except that our algorithm is on-line.

For any given string  $S$ , we can construct two strings  $S_1$  and  $S_2$  as follows:  $S_1$  is  $S$  with every parameter replaced by integer 0.  $S_2$  is  $S$  with every fixed symbol replaced by a single fixed symbol. We can construct the implicit suffix tree of  $\text{prefix}_i(S_1)$  by Ukkonen's algorithm and the s-suffix tree of  $\text{prefix}_i(S_2)$  by the above algorithm while constructing the implicit s-suffix tree for  $\text{prefix}_i(S_2)$ . Note that the construction of the suffix tree of  $S_1$  takes  $O(n \log |\Sigma|)$  time and that of the s-suffix tree of  $S_2$  takes  $O(n \log |\Pi|)$  time.

In any phase  $i$ , we can compute the length of the common prefix of s-encoded strings of  $\text{suffix}_j(\text{prefix}_i(S))$  and  $\text{suffix}_k(\text{prefix}_i(S))$  for any  $k < i$  and  $j < i$  as follows: According to [8], we can compute the lowest common ancestor of two nodes of a suffix tree in a constant time even while we are constructing the tree. Thus we can compute in a constant time the length of the common prefix of  $\text{suffix}_j(\text{prefix}_i(S_1))$  and  $\text{suffix}_k(\text{prefix}_i(S_1))$  for any  $k < i$  and  $j < i$ . We can also compute in a constant time the length of the common prefix of the s-encoded  $\text{suffix}_j(\text{prefix}_i(S_2))$  and  $\text{suffix}_k(\text{prefix}_i(S_2))$  for any  $k < i$  and  $j < i$ . The length of the common prefix of s-encoded strings of  $\text{suffix}_j(\text{prefix}_i(S))$  and  $\text{suffix}_k(\text{prefix}_i(S))$  for any  $k < i$  and  $j < i$  is the smaller of these two values.

By maintaining a set of edges that forms a zero-chain in a c-queue, we can speed up the scanning as follows: Consider a situation in which we encounter a zero-chain while scanning. First we find a leaf  $w$  that is a child of the deepest node in the zero-chain and is not a child of the target locus. Next, we compute the length of the common prefix length of the s-encoded strings of the the target

string and  $\sigma_w$ . Then we can find the deepest edge in the zero-chain that is an ancestor of the locus in  $O(\log |\Pi|)$  time. An normal zero-edge from a node can be accessed in a constant time. In this way, we can achieve an  $O(n(\log |\Sigma| + \log |\Pi|))$  computation time. For more details of this speeding-up technique, see [14].

## 4 Computational Experiments

Using the s-suffix tree of a string we can perform tasks such as the following:

- Given a long sequence of length  $n$  and some constant  $l$  and  $r$ , we can find a set of more than  $r$  substrings that s-match with each other and are longer than  $l$  in an  $O(n(\log |\Sigma| + \log |\Pi|) + T_{output})$  time, where  $T_{output}$  is the output size.
- Given more than one sequence, we can find the longest common s-encoded pattern of these sequences in  $O(n(\log |\Sigma| + \log |\Pi|) + T_{output})$  time, where  $T_{output}$  is the output size and  $n$  is the sum of the lengths of the input sequences.

Note that, if the size of the alphabet is constant, both of these tasks can be completed in a linear time. In this section, we describe experiments on RNA and DNA sequences, in which we constructed the s-suffix tree of DNA sequences, where  $\Sigma = \emptyset, \Pi = \{\text{A,U,G,C}\}$ , A is the complement of U and G is the complement of C. (In DNA sequences, T is present instead of U.)

We conducted experiments on three HIV (human immunodeficiency virus) RNA complete sequences: (A) a sequence of length 9719 (accession number: K03455), (B) a sequence of length 9748 (accession number: X01762) and (C) a sequence of length 8981 (accession number: AF067156). We also use four very long DNA sequences of E. coli, each of which has the same length, 1 Mbp = 1,000,000 bp. The length of the full genome sequence of E. coli is about 4.64 Mbp, and these four sequences are the following regions of the sequence: (D) 1 bp–1,000,000 bp, (E) 1,000,001 bp–2,000,000 bp (F) 2,000,001 bp–3,000,000 bp, and (G) 3,000,001 bp–4,000,000 bp.

First, we compare the size of the s-suffix tree with that of the normal suffix tree of the same sequences. Table 1 shows the numbers of nodes in the suffix trees and the s-suffix trees of the seven sequences. According to the table, the sizes of the s-suffix trees are slightly smaller than those of the normal suffix trees in all cases, but the numbers of nodes in them are almost the same regardless of the length of the sequence. For any sequence, both the number of nodes in the suffix tree and that of the s-suffix tree are about 1.6 to 1.7 times the length of the sequence. Thus we can say that the s-suffix trees are very compact and that it is as reasonable to build them as to build the normal suffix trees.

Consider that a structural pattern  $\alpha$  of length  $l$  appears  $r$  times, but any pattern that is constructed by extending it to the right, such as  $\alpha c$  ( $c \in (\Sigma \cup \Pi)$ ) appears less than  $r$  times. We call such a pattern  $\alpha$  a “maximal structural pattern.” We now give the experimental results of an experiment to find maximal structural patterns which are longer than  $l$  and repeated more than  $r$  times for

**Table 1.** Number of nodes in suffix trees and s-suffix trees

Sequence	(A)	(B)	(C)	(D)	(E)	(F)	(G)
Length	9719	9748	8981	1000000	1000000	1000000	1000000
Suffix Tree	16135	16217	14710	1640492	1635995	1638043	1638008
s-Suffix Tree	16033	16132	14666	1631525	1628821	1630104	1628923

**Table 2.** Examples of maximal structural pattern

(1)		(2)	
Position	Sequence	Position	Sequence
646095	CCCGCTTCGGCTTCA	371484	ACTGCGCCATGAAGATGAC
703617	GGGCCTTGGCGTTGA	884639	GACTATAAGCTGGTGCTGA
779110	TTTATGGTAATGGTC		
888469	TTTATCCTAACCTCG		

some given  $l$  and  $r$ . Table 2 shows two examples of maximal structural patterns found in *E. coli* sequence (D): (1) is a set of patterns of length 15 that appears four times, and (2) is a set of patterns of length 19 that appears 2 times in the sequence. Every sequence is different from the others, but these sequences s-match with each other.

Table 3 shows the number of maximal patterns whose lengths ( $l$ 's) are larger than some given length. In the table, a “normal pattern” means an ordinary string pattern that can be found with an ordinary suffix tree. Notice that the structural patterns includes the normal patterns. According to the table, we can see interesting facts such as that the proportion of normal patterns increases with the lengths of the patterns.

## 5 Concluding Remarks

We have proposed a new data structure called the structural suffix tree, or s-suffix tree for short. We also proposed an on-line  $O(n(\log |\Sigma| + \log |\Pi|))$  algorithm for constructing it, where  $\Sigma$  is an alphabet of fixed symbols and  $\Pi$  is an alphabet of parameters. This data structure enables an efficient search for frequent patterns of structures of RNA sequences or single-stranded DNA sequences. It also enables a common structure pattern to be efficiently found in more than one sequence. We also showed the practicality of our data structure and our algorithm by reporting computational experiments for finding structural patterns from RNA sequences of HIV and DNA sequences of *E. coli* using the s-suffix tree.

Several tasks remain for the future. Two sequences can have the same structure even if they do not have the same s-encoded string patterns. Furthermore, it is difficult to apply our algorithm to the problem of proteins, where the combinations are far more complicated. Thus we should strive to create more general

**Table 3.** Number of structural/normal patterns

(1) HIV RNA sequences			(2) E. coli sequences					
$l$	Pattern	(A)	(B)	(C)	(D)	(E)	(F)	(G)
$\geq 5$	Structural	5329	5061	4887	495371	499205	498728	497701
	Normal	1381	1147	1000	90968	85899	88681	90298
$\geq 10$	Structural	670	451	282	4723	4140	4466	4529
	Normal	479	363	126	2402	1728	2095	2147
$\geq 15$	Structural	336	123	4	330	106	192	192
	Normal	336	123	3	330	103	192	190

data structures and algorithms for structural pattern matching of biological sequences. Recently, Farach [9] introduced a linear-time suffix tree construction algorithm for strings of an integer alphabet  $\{1, \dots, n\}$ . It is an open problem whether or not such a linear time algorithm exists for constructing s-suffix trees or p-suffix trees.

## References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, “Design and Analysis of Algorithms,” Addison Wesley Publishing Co., Reading, Mass., 1974.
2. V. R. Akmaev, S. T. Kelley, and G. D. Stormo, “A Phylogenetic Approach to RNA Structure Prediction,” *Proc. 7th International Conference on Intelligent Systems for Molecular Biology (ISMB '99)*, 1999, pp. 10-17.
3. B. S. Baker, “A Program for Identifying Duplicated Code,” *Computing Science and Statistics*, Interface Foundation of North America, 1992, pp. 49-57.
4. B. S. Baker, “Parameterized Pattern Matching by Boyer-Moore-type Algorithms,” *Proc. 6th Annual ACM-SIAM Symp. Discrete Algorithms (SODA '95)*, 1995, pp. 541-550.
5. B. S. Baker, “Parameterized Pattern Matching: Algorithms and Applications,” *J. Comp. Syst. Sci.*, Vol. 52, No. 1, 1996, pp. 28-42.
6. B. S. Baker, “Parameterized Duplication in Strings: Algorithms and Application to Software Maintenance,” *SIAM J. Comput.*, Vol. 26, No. 5, 1997, pp. 1343-1362.
7. B. S. Baker, “Parameterized Diff,” *Proc. 10th ACM-SIAM Symp. Discrete Algorithms (SODA '99)*, 1999, pp. 854-855.
8. R. Cole and R. Hariharan, “Dynamic LCA Queries on Trees,” *Proc. 10th ACM-SIAM Symp. Discrete Algorithms (SODA '99)*, 1999, pp. 235-244.
9. M. Farach, “Optimal Suffix Tree Construction with Large Alphabets,” *Proc. 38th IEEE Symp. Foundations of Computer Science (FOCS '97)*, 1997, pp. 137-143.
10. D. Gusfield, “Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology,” *Cambridge University Press*, 1997.
11. D. Harel and R. R. Tarjan, “Fast Algorithms for Finding Nearest Common Ancestors,” *SIAM J. Computing*, Vol. 13, 1984, pp. 338-355.
12. H. P. Lenhof, K. Reinert, and M. Vingron, “A Polyhedral Approach to RNA Sequence Structure Alignment,” *Proc. 2nd Annual International Conference on Computational Molecular Biology (RECOMB '98)*, 1998, pp. 153-162.

13. R. B. Lyngso, M. Zuker, and C. N. S. Pedersen, "Internal Loops in RNA Secondary Structure Prediction," *Proc. 3rd Annual International Conference on Computational Molecular Biology (RECOMB '99)*, 1999, pp. 260-267.
14. S. R. Kosaraju, "Faster Algorithms for the Construction of Parameterized Suffix Trees," *Proc. 36th IEEE Symp. Foundations of Computer Science (FOCS '95)*, 1995, pp. 631-637.
15. E. M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," *J. ACM*, Vol. 23, 1976, pp. 262-272.
16. J. Setubal and J. Meidanis, *Introduction to Computational Molecular Biology*, PWS Pub. Co., Boston, 1997.
17. D. H. Turner, N. Sugimoto, and S. M. Freier, "RNA Structure Prediction," *Ann. Rev. Biophys. Chem.*, Vol. 17, 1988, pp. 167-192.
18. E. Ukkonen, "On-Line Construction of Suffix-Trees," *Algorithmica*, Vol. 14, 1995, pp. 249-60.
19. Z. Wang and K. Zhang, "Finding Common RNA Secondary Structures from RNA Sequences," *Proc. 4th Symposium on Combinatorial Pattern Matching*, Springer-Verlag LNCS 1645, 1999, pp. 258-269.
20. M. S. Waterman, *Introduction to Computational Biology*, Capman & Hall, London, 1995.
21. P. Weiner, "Linear Pattern Matching Algorithms," *Proc. 14th Symposium on Switching and Automata Theory*, 1973, pp. 1-11.

# Efficient Computation of All Longest Common Subsequences

Claus Rick

Institut für Informatik IV, Universität Bonn,  
Römerstr. 164, 53117 Bonn, GERMANY  
[rick@cs.uni-bonn.de](mailto:rick@cs.uni-bonn.de)

**Abstract.** Many efficient algorithms have been developed to compute the length of a longest common subsequence (LCS) between two strings. In general, an LCS is not unique but current methods only recover a single LCS. We investigate the problem of finding *all* longest common subsequences. A simple extension of the reconstruction method used by existing algorithms would seriously harm their time complexities. We present observations on a symmetry of the LCS problem which allow us to develop a general method to obtain a representation of all longest common subsequences while preserving the favorable time bounds of known algorithms.

## 1 Introduction

The problem of determining the similarity of two sequences  $A = a_1 a_2 \dots a_m$  and  $B = b_1 b_2 \dots b_n$ ,  $m \leq n$  over some finite alphabet  $\Sigma$  arises in many different areas of application, including one in the study of the evolution of long molecules. A widely accepted measure of similarity is the Levenshtein distance which can be evaluated by a dynamic programming algorithm in time  $\Theta(mn)$  [17]. A common subsequence is any sequence which can be obtained from both strings  $A$  and  $B$  by deleting zero or more (not necessarily adjacent) symbols. The length  $p$  of a longest common subsequence is closely related to the Levenshtein distance. In fact, it can be viewed as special case and thus a variation of the dynamic programming algorithm can be used to compute this value.

It was observed early that efficiency could be improved for typical applications by exploiting some structural properties of the LCS problem [9, 12]. The time complexities of such algorithms are parameterized by variables other than the sizes of the two input strings (for surveys see [3, 10]). For example, there are algorithms which perform well when the length of an LCS is short ( $O(pm)$  [1]) while preference would be given to an  $O(n(m - p))$  algorithm [13] when the length of an LCS is long. In Section 2 we will shortly review the paradigm underlying most of these algorithms. The primary goal of these methods is to calculate the length of an LCS as quick as possible.

In some applications, e.g. when building an alignment of two DNA sequences, one is also interested to see in which places the two sequences differ and where

they match. Alternatively, an edit script may be required which transforms  $A$  into  $B$  by insertions and deletions. All methods can easily be extended to recover a single longest common subsequence and this does not affect the asymptotic time complexity of the algorithms. However, a longest common subsequence is not unique and the (arbitrary) LCS reconstructed is primarily due to the implementation of a particular algorithm. Thus it may not reveal an expected relationship between the two sequences which another LCS would be able to indicate. Therefore it is of interest to compute a representation which highlights the structure of *all* longest common subsequences and from which each single LCS may be recovered easily. So far, this may only be achieved by the dynamic programming algorithm and (partially) by the construction proposed in [2].

As we will argue in Section 3, simple extension of the classical method (trace back) to recover an LCS used by all the efficient algorithms will seriously harm their time complexities. In Section 4 we give a characterization of LCSs which is based on a symmetry of the LCS problem and in Section 5 we show how to use it to maintain the original time complexities of the algorithms while computing a suitable representation of all longest common subsequences.

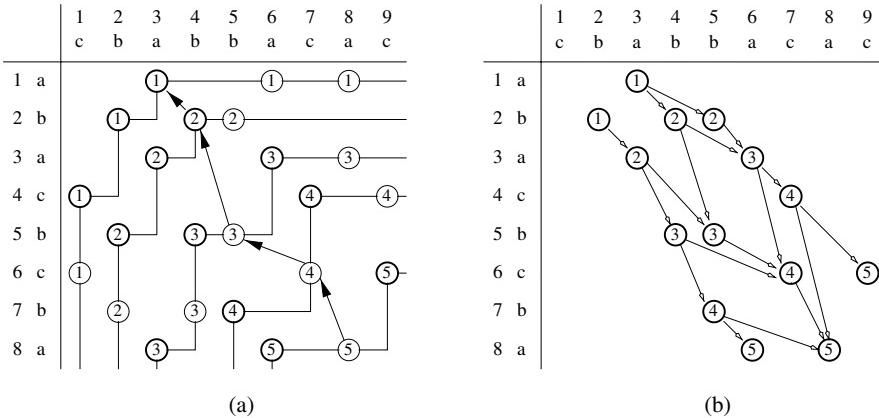
## 2 Matches and Contours

It is common to describe the LCS problem in the following way. An ordered pair  $(i, j)$ ,  $1 \leq i \leq m$ ,  $1 \leq j \leq n$  is called a *match* if  $a_i = b_j$ . The set  $M$  of all matches can be represented by a *matching matrix* of size  $m \times n$  in which each match is identified by a circle. Two matches  $(i, j)$  and  $(i', j')$  may be part of the same common subsequence if and only if  $i < i' \wedge j < j'$  or  $i' < i \wedge j' < j$ . A sequence  $S \subseteq M$  of matches that is strictly increasing in both components is called a *chain*. The LCS problem can now be viewed as finding a longest chain. It is solved by employing a technique called sparse dynamic programming [7] which rests on some structural properties of the LCS problem.

Let  $|LCS(A, B)|$  denote the length of an LCS between strings  $A$  and  $B$  and let  $A_i = a_1 \dots a_i$ ,  $0 \leq i \leq m$  denote the length  $i$  prefix of  $A$ . For a match  $(i, j)$  we say that it is of *rank*  $k$  if the length of a longest chain ending at  $(i, j)$  is  $k$ . We can collect matches of the same rank  $k$  in classes

$$C_k = \{(i, j) \in M : |LCS(A_i, B_j)| = k\}.$$

Thus,  $M$  can be partitioned into classes  $C_1, C_2, \dots, C_p$ , each class containing matches of the same rank. It is well known that these classes exhibit a special structure in the matching matrix. If sorted in increasing order with respect to the first component and in decreasing order with respect to the second component, matches belonging to the same class shift from right to left, and they form so-called *contours* when connected by lines as shown in Figure 1(a). Contours of different classes may never cross or touch, and the contour of each class divides the matrix into a top/left part and a bottom/right part. Each contour can be completely specified by *dominant matches*, i.e. those matches  $(i, j)$  in a class for which there is no other match  $(i', j')$  in the same class with  $i' = i \wedge j' < j$  or



**Fig. 1.** Matches and Contours (a), LCSs-graph (b).

$i' < i \wedge j' = j$ . We use  $D_k \subseteq C_k$  to denote the dominant matches of rank  $k$ . They are located in the top/left corners of a contour and they are indicated by bold circles in Figure 1(a). On the other hand each match in  $C_k \setminus D_k$  is dominated by some match in  $D_k$ . Thus there will always be an LCS consisting only of dominant matches. For more background on this see for example [14].

Speedup was gained for many algorithms by concentrating on the computation of dominant matches and by exploiting the above properties in clever ways [13, 11, 1, 6, 15]. For example, the algorithm introduced in [15] has time complexity  $O(n|\Sigma| + \min\{pm, p(n-p)\})$  where time  $O(n|\Sigma|)$  is used for some preprocessing (solving the so-called string identification problem) and time  $O(\min\{pm, p(n-p)\})$  is used to determine the dominant matches. Our goal will be to maintain such favorable time complexities while computing a representation of all LCSs. To this end it is important to note that the time complexity of the main processing stage is an upper bound on the number of dominant matches since each dominant match is touched at least once by these algorithms [14]. On the other hand it is known that  $p \leq d \leq r \leq mn$ . In particular there is no fixed correlation between the total number of matches,  $r$ , and the number of dominant matches,  $d$ . There are instances where the latter can be very small while the former essentially remains quadratic. E.g., consider strings over a small alphabet where each symbol occurs the same number of times and which have an LCS close to the string length.

### 3 Representing All LCSs

A single LCS may be recovered by storing a pointer with each dominant match which points to one of his direct predecessors in a chain. Starting with any dominant match of highest rank we can trace an LCS in time  $O(p)$  by simply following these pointers. This is the classical way to reconstruct an optimal

solution in dynamic programming methods. But in general an LCS is not unique. Consider the two strings  $A = abacbcba$  and  $B = cbabbacac$ . Their structure is shown in Figure 1(a). There are five different character sequences which all form an LCS, namely  $abacc$ ,  $abaca$ ,  $abbca$ ,  $babca$ , and  $babba$ . Further, for each character sequence there may exist different *embeddings*, i.e. positions in the two strings to which the characters of an LCS map. E.g., the sequence  $abacc$  may correspond to the sequence of matches  $(1, 3), (2, 4), (3, 6), (4, 7), (6, 9)$  or to  $(1, 3), (2, 5), (3, 6), (4, 7), (6, 9)$ . A *canonical embedding* of a fixed LCS is an embedding where each character, starting from the beginning of the LCS, is assigned matching positions in both sequences as small as possible. So the first sequence of matches above is a canonical embedding while the second is not.

In [2] a directed acyclic subsequence graph (DASG) is defined as a finite automaton recognizing all subsequences of a string. The generalization for two strings is a representation of the canonical embeddings of all LCSs and can be built in time  $O(n \log n + r)$ . The only representation which includes all embeddings of all LCSs is the dynamic programming matrix augmented with appropriate back-pointers which takes time and space  $\Theta(mn)$  to construct.

For each match  $(i, j)$  define  $CS(i, j)$  to be the maximum length of a common subsequence of  $A$  and  $B$  containing  $(i, j)$ , i.e.

$$CS(i, j) := \max\{|S| : S \text{ is a chain and } (i, j) \in S\}.$$

We will now define the *LCSs-Graph* which seems to be the most appropriate structure to represent all LCSs, including different embeddings.

**Definition 1 (LCSs-Graph).** *The LCSs-Graph of two Sequences  $A$  and  $B$  which have LCSs of length  $p$  is the directed acyclic Graph  $G = (V, E)$ , where*

1.  $V = V_1 \dot{\cup} V_2 \dot{\cup} \dots \dot{\cup} V_p$ ,
2.  $V_k = \{(i, j) \mid (i, j) \in C_k \wedge CS(i, j) = p\}$ ,
3.  $E = \{[(i, j), (i', j')] \mid (i, j) \in V_k, (i', j') \in V_{k+1}, i < i', j < j'\}$ .

The LCSs-graph for our example of Figure 1(a) is given in Figure 1(b). Note that edges are easily derived from the nodes and so they need not be given explicitly. Thus, our primary concern will be the efficient computation of nodes. The goal is to construct  $G$  in time  $O(T + |G|)$  where  $T$  is the time of any algorithm which determines the dominant matches.

An important fact to note is that in general dominant matches are not sufficient to represent all LCSs. It is easily checked that the LCS  $abbca$ , indicated by back-arrows in Figure 1(a), can not be represented solely by dominant matches. On the other hand we don't (yet) have a criterion to decide which non-dominant matches are necessary for the construction of some LCS and which are not.

Therefore, if we decide to reconstruct all LCSs using the classical back-pointer-approach there seems to be no way to avoid the creation of a separate node for every match. Further, with each match we may have to store several back-pointers. So this method will take time  $\Omega(r)$ . Such a time bound is disastrous for all algorithms concentrating on the computation of dominant matches. Their time complexity usually gives an upper bound on the number of dominant matches but not on the total number of matches (see Section 2).

## 4 A Characterization Based on Structural Symmetries

Our goal in this section is to establish a characterization of the nodes of the LCSS-graph which allows us to keep our attention restricted to dominant matches. This will support the development of an efficient construction to be given in the next section.

We exploit a symmetry of the LCS-problem which already has been used fruitfully in connection with linear space computations [8]. Usually, the input strings are considered in the typical reading direction from left to right (or front to rear). This, however, is an arbitrary decision and in fact any algorithm can be easily modified to work in the opposite direction (equivalently, the original algorithm may be applied to the reversed input strings). We will note some general facts on the relationship of contours computed in the usual way, called *forward contours*, and those computed by considering the strings in the opposite direction, called *backward contours*. Although the number of contours is identical we note that forward contours and backward contours might look very different (see Fig. 1(a) and Fig. 2(a)). In particular dominant matches on forward contours need not be dominant matches on backward contours and vice versa. Since contours partition the set of matches there is a unique forward contour and a unique backward contour for each match. As before we use  $C_k$  and  $D_k$  to denote matches and dominant matches on the  $k$ -th forward contour, respectively. Likewise

$$\widehat{C}_k = \{(i, j) \in M : |LCS(A^i, B^j)| = k\}$$

is the set of matches on backward contour  $k$  where  $A^i = a_i \dots a_m$ ,  $1 \leq i \leq m+1$  denotes the length  $m-i+1$  suffix of  $A$ . The set  $\widehat{D}_k \subseteq \widehat{C}_k$  of dominant matches on backward contour  $k$  is formally defined by

$$\widehat{D}_k := \{(i, j) \in \widehat{C}_k \mid \nexists (i', j') \in \widehat{C}_k : (i' = i \wedge j' > j) \vee (i' > i \wedge j' = j)\}.$$

The fundamental observation, which forms the basis of our approach, is that the value  $CS(i, j)$  may be calculated very easily from the unique forward contour and the unique backward contour a match  $(i, j)$  belongs to.

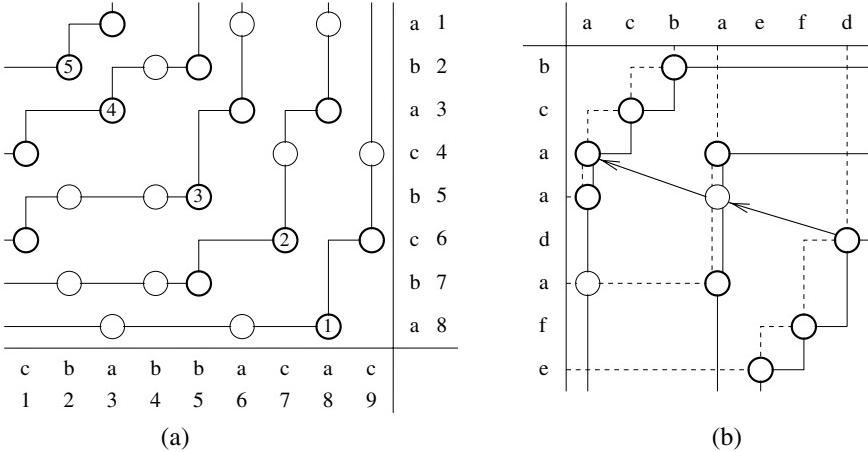
**Lemma 1.** *For each match  $(i, j)$  the following holds:*

$$(i, j) \in C_k \cap \widehat{C}_{k'} \implies CS(i, j) = k + k' - 1.$$

*Proof.* Since  $(i, j) \in C_k$  there is a chain of length  $k$  ending at  $(i, j)$  and from  $(i, j) \in \widehat{C}_{k'}$  we can conclude that there exists a chain of length  $k'$  starting at  $(i, j)$ . Joining these chains at  $(i, j)$  gives a chain of length  $k + k' - 1$ .

Assume there is a common subsequence of length  $> k + k' - 1$  containing  $(i, j)$ . Then there would be a chain of length  $> k$  ending at  $(i, j)$  or there would be a chain of length  $> k'$  starting at  $(i, j)$ . But this would contradict  $(i, j) \in C_k$  or  $(i, j) \in \widehat{C}_{k'}$ , respectively.  $\square$

This fact immediately implies the following characterization of the nodes  $V = V_1 \dot{\cup} V_2 \dot{\cup} \dots \dot{\cup} V_p$  of our LCSS-graph  $G = (V, E)$ .



**Fig. 2.** Backward contours (a) and essential non-dominant matches (b).

**Corollary 1.** Let  $p$  be the length of an LCS of two input sequences  $A$  and  $B$ . Then

$$(i, j) \in V_k \iff (i, j) \in C_k \cap \widehat{C_{k'}}, \quad k' = p - k + 1.$$

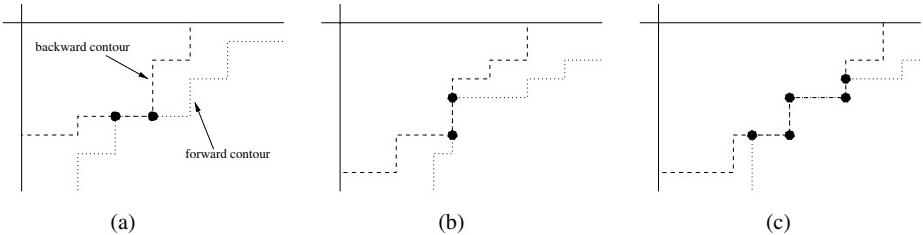
We call a forward contour and a backward contour *complementary* if their ranks add up to  $p + 1$ . The major drawback of Corollary 1 is that the given characterization still relies on *all* matches on the various contours. With regard to the efficient LCS-algorithms it would be helpful to have a characterization solely in terms of dominant matches. On the other hand, as can be seen by the example given in Figure 2(b), there may be essential matches for the LCSs-graph which are neither dominant on a forward contour (solid lines) nor dominant on a backward contour (dashed lines). The LCS  $aad$  may not be generated without using the non-dominant match  $(4, 4)$ .

Our initial observation is that two complementary contours may touch but never cross each other. Further, the forward contour is to the bottom/right of the backward contour and both contours have a least one match in common.

**Lemma 2.** Let  $C_k$  and  $\widehat{C_{k'}}$  be two complementary contours, i.e.  $p = k + k' - 1$  where  $p$  is the length of a LCS. Then it holds that

1.  $\nexists ((i, j) \in C_k \wedge (i', j') \in \widehat{C_{k'}} : i < i' \wedge j < j',$
2.  $\exists (i, j) \in C_k \cap \widehat{C_{k'}}.$

*Proof.* Assume there are matches  $(i, j) \in C_k$  and  $(i', j') \in \widehat{C_{k'}}$  such that  $i < i'$  and  $j < j'$ . Then  $(i', j') \in C_l$ ,  $l > k$  (and  $(i, j) \in \widehat{C_{l'}}$ ,  $l' > k'$ ). So we could form a chain of length  $l + k' - 1 > k + k' - 1 = p$ . A contradiction. If there would be no match  $(i, j) \in C_k \cap \widehat{C_{k'}}$  then, by Corollary 1,  $V_k = \emptyset$  and thus no LCS of length  $p$  could exist.  $\square$



**Fig. 3.** The shape of common parts of complementary contours.

We can now give the desired characterization. Informally, it states that common parts of complementary contours can be completely specified by means of pairs of *dominant* matches on both contours.

**Lemma 3.** *Let  $C_k$  be a forward contour and let  $\widehat{C_{k'}}$ ,  $k' = p - k + 1$ , be a complementary backward contour. Then it holds:*

$$(i, j) \in V_k \iff \exists (x, y) \in D_k, \exists (x', y') \in \widehat{D_{k'}} : \\ (x = i = x' \wedge y \leq j \leq y') \vee (x \leq i \leq x' \wedge y = j = y').$$

*Proof.* First we prove that a match  $(i, j)$  according to the given characterization in fact belongs to  $V_k$ . We show that  $(i, j) \in C_k \cap \widehat{C_{k'}}$ . Consider the case  $x \leq i \leq x' \wedge y = j = y'$  (the case  $x = i = x' \wedge y \leq j \leq y'$  is symmetric). We claim that these prerequisites imply  $(x', y') \in C_k$ , and therefore  $(i, j) \in C_k$ . Assume  $(x', y') \notin C_k$ . Since  $y = y'$ , contour  $C_k$  would then have to take a left turn before reaching row  $x'$ . But this means there has to be a dominant match  $(x'', y'') \in D_k$  such that  $x'' < x' \wedge y'' < y'$ , contradicting Lemma 2. A similar argument shows that  $(x, y) \in \widehat{C_{k'}}$  and hence  $(i, j) \in \widehat{C_{k'}}$ .

In order to prove the other direction we have to show that each match  $(i, j) \in V_k$  can be characterized in the claimed way. To this end we will show that all common parts of two complementary contours, containing all matches in  $C_k \cap \widehat{C_{k'}}$ , can be split into horizontal and vertical pieces such that each piece will be bordered by a dominant match  $(x, y) \in D_k$  and by a dominant match  $(x', y') \in \widehat{D_{k'}}$ . From Lemma 2 we know that complementary contours may only touch but never cross each other and that the backward contour is to the top/left of the forward contour. Now follow both complementary contours from top/right to bottom/left up to the first common point  $(i, j)$ .

The main observation is that  $(i, j) \in D_k \vee (i, j) \in \widehat{D_{k'}}$ . There is only one possibility how both contours may join, namely when a vertical piece of the backward contour meets a horizontal piece of the forward contour. Since the contours may not cross each other there must be a bend in this point. If the bend is on the backward contour  $(i, j) \in \widehat{D_{k'}}$  (see Fig. 3 (a)) and if the bend is on the forward contour  $(i, j) \in D_k$  (see Fig. 3 (b)). If there is a bend on both contours  $(i, j) \in D_k \cap \widehat{D_{k'}}$  and the contours only touch in this point. In the former cases both contours may have some path in common. If this path includes

several bends (see Fig. 3 (c)) there must be a dominant match at each corner of these bends alternately from forward and backward contours. Finally, in some point the contours may spread apart again and there is a dominant match at this point too for reasons of symmetry.

Thus, each common horizontal piece is bordered by dominant matches  $(x, y) \in D_k$  and  $(x', y') \in \widehat{D}_{k'}$  such that  $x \leq x' \wedge y = y'$  and each common vertical piece is similarly bordered by dominant matches such that  $x = x' \wedge y \leq y'$ .  $\square$

Note that this characterization allows us to give a compact description of the possibly much greater LCSs-graph. For each pair of complementary contours we only have to record pairs of dominant matches which sandwich common parts of both contours. I.e., it is sufficient to compute the sets  $V'_k \subseteq V_k$  where

$$V'_k := (D_k \cap \widehat{C}_{k'}) \cup (\widehat{D}_{k'} \cap C_k), \quad k = 1, \dots, p \text{ and } k' = p + 1 - k.$$

Let  $V' = \cup_{k=1}^p V'_k$ . Such a compact description may be helpful, if

- we need a space-efficient way to store all optimal solutions which allows the recovery of an explicit representation quickly if needed (see below);
- further computations are to be done on all optimal solutions. It may be more efficient to do such computations on the compact representation [4].

We can also use the sets  $V'_k$  to specify two distinguished LCSs. Consider the match  $r_k \in V'_k$  which is to the top/right of any other match in  $V'_k$  and similarly let  $s_k \in V'_k$  denote the match which is to the bottom/left of any other match in  $V'_k$ . Then, figurally speaking, all LCSs will be located somewhere between the two outer LCSs  $R = r_1 r_2 \dots r_p$  and  $S = s_1 s_2 \dots s_p$ . This kind of information might be useful in a learning algorithm for the LCS-problem presented lately [5]. It can also be combined with a recent proposal for linear-space implementations [16] to devise a space-saving variation of the following method to construct the LCSs-graph.

## 5 An Efficient Construction

Based on Lemma 3 we can now develop a general method to determine the nodes of the LCSs-graph. In a first step, simply compute the forward contours as well as the backward contours by any method of your choice. Then, for each pair of complementary contours, find the matches belonging to both contours. In order to perform this second step efficiently we assume that contours are given by linked lists, one for each forward contour and one for each backward contour, containing the (dominant) matches on the respective contours in sorted order, i.e. from top/right to bottom/left. Such lists may easily be generated by any algorithm during the first step of the computation.

Let  $L$  and  $\widehat{L}$  be two such lists corresponding to complementary contours. If these lists would contain *all* matches then our task would be to find identical matches occurring on both lists. In view of Lemma 2 this could be done by a

simple appropriate scan which takes time proportional to the length of the two lists. For reasons of efficiency, however, we may only assume that the given lists contain the *dominant* matches of the respective contours. Lemma 3 showed that knowing those dominant matches on complementary contours which belong to both contours is sufficient to specify all nodes of the LCSSs-graph  $G = (V, E)$ . Since a dominant match on a forward contour need not be a dominant match on the complementary backward contour, and vice versa, we can no longer use a simple scan looking for identical matches on two complementary lists. But from Lemma 3 we know that if a dominant match actually belongs to  $V$  then there must exist a dominant match on the complementary contour located in the same row or column. Using this fact our desired dominant matches can still be found in time proportional to the length of the two involved complementary lists by scanning them as shown by the procedure “LCS-Merge” given in Figure 4. We assume that the two input lists are sorted and that the end of the lists will be marked by a sentinel  $(\infty, \infty)$ . In order to avoid duplicates in the output list we assume that a match is only appended to the list by the operator  $\cdot$  if it is not identical to the current last element of the list.

**Procedure** LCS-Merge

**Input:**  $L$ , a sorted list of dominant matches on a forward contour.  
 $\widehat{L}$ , a sorted list of dominant matches on a complementary backward contour.

**Output:**  $L_G$ , a sorted list of those dominant matches from the two input lists which may be part of an LCS.

**Method:**

1.  $(x, y) \leftarrow L.\text{first}$
2.  $(x', y') \leftarrow \widehat{L}.\text{first}$
3. **while**  $(x, y) \neq (\infty, \infty)$  **and**  $(x', y') \neq (\infty, \infty)$  **do**
4.   **case**
5.      $y' < y$  :  $(x, y) \leftarrow L.\text{next}$
6.      $x' < x$  :  $(x', y') \leftarrow \widehat{L}.\text{next}$
7.      $x' = x \wedge y' \geq y$  :  $L_G \leftarrow L_G \cdot (x', y') \cdot (x, y)$ ;  $(x', y') \leftarrow \widehat{L}.\text{next}$
8.      $x' \geq x \wedge y' = y$  :  $L_G \leftarrow L_G \cdot (x, y) \cdot (x', y')$ ;  $(x, y) \leftarrow L.\text{next}$
9.   **end**

**Fig. 4.** Procedure identifying dominant matches on complementary contours.

**Theorem 1.** *A compact representation of all LCSSs, i.e. the node set  $V' \subseteq V$ , can be computed in time  $O(T)$  and space  $O(S)$  where  $T$  and  $S$  is the time and space, respectively, of any algorithm creating all dominant matches.*

*Proof.* The chosen algorithm will be invoked two times to determine the dominant matches on forward contours and backward contours, respectively. Using the procedure “LCS-Merge” to identify common parts of complementary contours takes time proportional to the total number of dominant matches which is upper bounded by  $T$ .

$O(d)$  space will be occupied by the lists of dominant matches where  $d$  is the total number of dominant matches on forward contours and backward contours. Depending on the chosen algorithm we may need some additional space to store information gained in a preprocessing stage solving the so-called string identification problem.  $\square$

If needed the LCSs-graph can be constructed explicitly from our lists of dominant matches  $V'_k$  in time proportional to the size of the graph. In order to determine the non-dominant matches still missing we just have to consider two succeeding dominant matches on these sorted lists. If they have a common component they are the endpoints of an interval which may contain additional matches to be included. Using two standard lookup-tables which contain the next occurrence of a symbol  $\sigma \in \Sigma$  to the right of a given position in strings  $A$  and  $B$ , respectively, we can identify each such match in constant time. These tables, which take  $O(|\Sigma| \cdot n)$  time and space to construct, are required anyway by many algorithms computing dominant matches. There is also an  $O(n)$  time and space variation of these tables which provides the desired information in  $O(\log |\Sigma|)$  time per query II.

Edges can be determined according to the definition of the LCSs-graph by scanning sorted neighboring node lists  $V_k$  and  $V_{k+1}$  as follows. We consider each match on  $V_k$  in succession. Let  $(x, y)$  be the first match on  $V_k$ . Scanning  $V_{k+1}$ , we find the first match  $(x', y')$  such that  $x < x' \wedge y < y'$ . A pointer  $P$  to this position on  $V_{k+1}$  is saved and we insert the edge  $([x, y], [x', y'])$ . Proceeding on  $V_{k+1}$ , we continue to insert edges leaving  $(x, y)$  as long as  $x < x' \wedge y < y'$  holds. Then we consider the next match on  $V_k$ , starting the scan on  $V_{k+1}$  at the position indicated by  $P$ . Note that in general matches on  $V_{k+1}$  will be considered several times. We distinguish two cases depending on whether an edge is inserted when considering a match on  $V_{k+1}$  or not. In the former case we can assign the cost to the the edge inserted. There are two cases which do not lead to the insertion of an edge:

1. searching for the first match on  $V_{k+1}$  which is the endpoint of an edge leaving the current match on  $V_k$ : in this case the pointer  $P$  ensures that this occurs at most once for each match on  $V_{k+1}$ .
2. reaching the first match on  $V_{k+1}$  which no longer is an endpoint of an edge leaving the current match on  $V_k$ : this happens at most once for each match on  $V_k$ .

Thus, we have shown the following theorem.

**Theorem 2.** *The LCSs-graph  $G = (V, E)$  can be constructed explicitly in time and space  $O(n|\Sigma| + |V| + |E|)$  from the compact representation  $V'$ .*

If one is not interested in different embeddings of the same character sequence of an LCS we can also construct a reduced graph  $\tilde{G} = (\tilde{V}, \tilde{E})$  from  $G$  which only contains a single (canonical) embedding for each possible character sequence of an LCS. This is done in time  $O(|G|)$  via an appropriate breadth first search on  $G$  which eliminates all edges and nodes not suitable for a canonical embedding.

## 6 Conclusion

Using a symmetry of the LCS problem we gave an exact characterization of all matches possibly occurring on an LCS. From this we developed a general method to compute a compact representation of all LCSs while maintaining the favorable time complexities of known efficient algorithms for determining the length of an LCS. A more suitable representation, the LCSs-graph  $G$ , can be constructed from the compact representation in time proportional to the size of  $G$ .

It would be interesting to see whether a characterization similar to Lemma 3 could be given for more than two input sequences. Another open question concerns the number  $d$  of dominant matches on forward contours and the number  $\hat{d}$  of dominant matches on backward contours. Does the structure of the LCS problem allow to establish an upper bound on  $|d - \hat{d}|$  which shows that these two values may not differ very much?

## References

- [1] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.
- [2] R. A. Baeza-Yates. Searching subsequences. *Theoretical Computer Science*, 78(2):363–376, January 1991.
- [3] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms (extended abstract). Technical report, Dept. of Computer Science, University of Turku, Finland, April 2000. manuscript.
- [4] N. Blum. Speeding up dynamic programming without omitting any optimal solution and some applications in molecular biology. *Journal of Algorithms*, to appear, 2000.
- [5] E. Breimer, M. Goldberg, and D. Lim. A learning algorithm for the longest common subsequence problem. In *Proceedings of the 2nd Algorithm Engineering and Experiments*, pages 147–156, San Francisco, CA, 2000.
- [6] Francis Y. L. Chin and C. K. Poon. A fast algorithm for computing longest common subsequences of small alphabet size. *Journal of Information Processing*, 13(4):463–469, 1990.
- [7] David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano. Sparse dynamic programming I: Linear cost functions. *Journal of the ACM*, 39(3):519–545, July 1992.
- [8] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, June 1975.
- [9] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *Jrnl. A.C.M.*, 24(4):664–675, 1977.

- [10] D. S. Hirschberg. Serial computations of Levenshtein distances. In A. Apostolico and Z. Galil, editors, *Pattern matching algorithms*, chapter 4, pages 123–141. Oxford University Press, 1997.
- [11] W. J. Hsu and M. W. Du. New algorithms for the LCS problem. *Journal of Computer and System Sciences*, 29:133–152, 1984.
- [12] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, May 1977.
- [13] N. Nakatsu, Y. Kambayashi, and S. Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18:171–179, 1982.
- [14] Claus Rick. A new flexible algorithm for the longest common subsequence problem. *Nordic Journal of Computing*, 2(4):444–461, Winter 1995.
- [15] Claus Rick. A new flexible algorithm for the longest common subsequence problem. In Zvi Galil and Esko Ukkonen, editors, *Combinatorial Pattern Matching, 6th Annual Symposium*, volume 937 of *Lecture Notes in Computer Science*, pages 340–351, Espoo, Finland, 5–7 July 1995. Springer.
- [16] Claus Rick. Simple and fast linear space computation of longest common subsequences. *submitted for publication*, 2000.
- [17] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, January 1974.

# A Blocked All-Pairs Shortest-Paths Algorithm

Gayathri Venkataraman, Sartaj Sahni, and Srabani Mukhopadhyaya

Department of Computer and Information Science and Engineering  
University of Florida, Gainesville, FL 32611  
{gvenkata, sahni, srabani}@cise.ufl.edu

**Abstract.** We propose a blocked version of Floyd's all-pairs shortest-paths algorithm. The blocked algorithm makes better utilization of cache than does Floyd's original algorithm. Experiments indicate that the blocked algorithm delivers a speedup (relative to the unblocked Floyd's algorithm) between 1.6 and 1.9 on a Sun Ultra Enterprise 4000/5000 for graphs that have between 480 and 3200 vertices. The measured speedup on an SGI O2 for graphs with between 240 and 1200 vertices is between 1.6 and 2.

**Keywords:** All pairs shortest paths, blocking, cache, speedup.

## 1 Introduction

Traditionally, algorithms are developed, analyzed, and optimized for the RAM computer model in which a computer has a single uniformly accessible memory [11]. Contemporary computers, however, have multiple levels of memory and the memory access time varies significantly from one memory level to the next. For example, contemporary Sun and SGI workstations have an L1 cache, an L2 cache, and a main memory. The L1 cache in a Sun Ultra Enterprise 4000/5000 is 16 KB, the L2 cache is 4 MB, and main memory is in excess of 100 MB. Additionally, a contemporary computer has a limited number of registers—ten to twenty. Typically, it takes 1 cycle to access data from L1 cache. When the desired data is not in L1 cache, we experience an L1 *miss* and the data is brought from L2 cache to L1 cache using 6 to 10 cycles. If the desired data is not in L2 cache either, then we experience an L2 miss and data is fetched from main memory into L2 cache at a cost of (say) 50 cycles, and from there to L1 cache. We can reduce run time by organizing our computations so as to minimize the number of L1 and L2 cache misses.

Although several theoretical models for computers with multiple-level memories have been proposed [32,7], these models have not found wide application, and most of the work in the area of performance enhancement via cache optimization has been experimentally oriented. Trace driven simulators have been used to study the cache performance of a specific program running on a specific computer, determine the portions of the code or the data structures that result in a large fraction of the cache misses, and then optimize these code segments and/or data structures. Trace driven simulations have also been used to develop

analytical models of cache behavior. See [4,15,19,22,23,24], for example, for some ways in which trace driven simulators have been used in cache performance enhancement studies.

La Marca and Ladner [13] develop a model for a single-level direct-mapped cache. They use this model to analyze the performance of binary heaps and cache-aligned  $d$ -heaps. LaMarca and Ladner [14] optimize the cache performance of several sorting methods. Their cache optimized heapsort and mergesort codes achieve a speedup of 1.85 and 1.38, respectively, when sorting 1,000,000 uniformly distributed integers on a Sprac 10 processor. Lam, Rothberg, and Wolf [12] have considered the cache performance of a blocked matrix multiply code relative to a traditional matrix multiply code. They report a speedup of 4.3 for their blocked matrix multiply code for a matrix of size 300. Sulatycke and Ghose [21] and Stewart [20] have also studied the cache performance of various matrix multiplication algorithms. Stewart [20] reports that the best way to multiply the matrices  $A$  and  $B$  is to first transpose  $B$  and then use the classical three loop algorithm on  $A$  and  $B^T$ . He further reports that by simply reordering the loops from the traditional  $ijk$  order to an  $ikj$  order (i.e., interchange the second and third **for** loops in the traditional code) the code performance is about the same as when square blocks (as used in [12] are used); row blocks yield superior speedup than column blocks and  $ikj$  ordering. Note that the transpose method,  $ikj$  ordering, square blocking, and row blocking deliver speedup relative to the traditional  $ijk$  code by reducing cache misses. Stewart [20] reports a speedup of 2.7 for the transpose method relative to the  $ijk$  code; both codes were written in C and compiled using maximum compiler optimization; the matrix size was 1200, and the code was run on a SUN Ultra Enterprise 4000/5000 computer.

Al-Furaih and Ranka [5,6] have studied cache optimization methods for sorting and unstructured iterative computations.

In this paper we propose a blocked formulation of Floyd's dynamic programming algorithm to find the lengths of the shortest paths between all pairs of vertices in a graph [1]. Blocked (or tiled) computation methods have been used before (for example, [16,10,25,12,9,1]). Our blocked algorithm provides a speedup (relative to the unblocked algorithm) between 1.6 and 1.9 on a Sun Ultra Enterprise 4000/5000 for graphs that have between 480 and 3200 vertices. The measured speedup on an SGI O2 for graphs with between 240 and 1200 vertices is between 1.6 and 2. These speedups are comparable to the speedups cited above for cache-optimized sorting and matrix multiplication codes on Sun platforms.

In Section 2 we give Floyd's all-pairs shortest-paths algorithm. Section 3 analyzes the potential speedup benefits from reorganizing Floyd's algorithm to make better use of cache. This analysis uses data gathered using the cache simulation tool *Shade* [17]. Our blocked version of Floyd's algorithm and a correctness proof are given in Section 4. Section 5 gives measured speedup results for our blocked algorithm.

## 2 Floyd's All-Pairs Shortest-Paths Algorithm

Let  $G = (V, E)$  be a directed graph with  $n$  vertices. Let  $\text{cost}$  be the cost adjacency matrix for  $G$ . So  $\text{cost}(i, i) = 0$ ,  $1 \leq i \leq n$ ;  $\text{cost}(i, j)$  is the length (or cost) of edge  $(i, j)$  if  $(i, j) \in E(G)$  and  $\text{cost}(i, j) = \infty$  if  $i \neq j$  and  $(i, j) \notin E(G)$ .

In the all-pairs shortest-paths problem we are to determine a matrix  $A$  such that  $A(i, j)$  is the length of a shortest path from  $i$  to  $j$ . When  $G$  has no cycle whose length (cost) is less than 0, the matrix  $A$  may be computed using dynamic programming [11]. Let  $A^k(i, j)$  be the length of a shortest path from  $i$  to  $j$  under the constraint that the path contain no intermediate vertex whose index is more than  $k$ . It is easy to see that  $A(i, j) = A^n(i, j)$ . When  $G$  has no cycle with negative length, the following dynamic programming recurrence is valid:

$$A^0(i, j) = \text{cost}(i, j) \quad (1)$$

$$A^k(i, j) = \min\{A^{k-1}(i, j), A^{k-1}(i, k) + A^{k-1}(k, j)\}, k \geq 1 \quad (2)$$

Equations 1 and 2 lead to the algorithm of Figure 1 to compute  $A$ . This algorithm is known as Floyd's algorithm. It may be shown [11] that **AllPairs** computes  $A^k(i, j) = A[i][j]$  in iteration  $k$  of the outermost **for** loop.

```

function AllPairs(int A, int n)
{ // A[i][j] = cost(i,j) initially
  // A[i][j] equals length of shortest
  // i to j path on termination
  for (k = 1; k <= n; k++)
    for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
        A[i][j] = min(A[i][j],
                      A[i][k] + A[k][j]);
}

```

**Fig. 1.** Floyd's shortest-paths algorithm

## 3 Upper Bound on Attainable Speedup

We compute an upper bound on the maximum speedup attainable by rearranging the computation of Figure 1 so as to optimize cache usage. In computing this bound we assume that any rearrangement of the computation will not decrease the number of accesses made to the elements of the array  $A$ .

We first obtain an equation to estimate the execution/run time of Floyd's algorithm of Figure 1. The execution time of a program is given by the following

equation [18]:

$$\begin{aligned} \text{execution time} &= (\text{CPU clock cycles} + \\ &\quad \text{memory stall cycles}) \\ &\quad \times \text{clock cycle time} \end{aligned} \quad (3)$$

where *memory stall cycles* is the number of cycles the CPU spends waiting for a memory reference to complete. The following equations are also from [18].

$$\text{CPU clock cycles} = \text{CPI} \times \text{IC} \quad (4)$$

$$\begin{aligned} \text{memory stall cycles} &= \text{number of L1 misses} \times \\ &\quad \text{L1 miss penalty} \end{aligned} \quad (5)$$

$$\text{number of L1 misses} = \text{IC} * \text{L1 misses per instruction} \quad (6)$$

$$\begin{aligned} \text{L1 misses per instruction} &= \text{memory references} \\ &\quad \text{per instruction} \\ &\quad \times \text{L1 miss rate} \end{aligned} \quad (7)$$

where *IC* is the *instruction count*, *CPI* is the *clock cycles per instruction*, *L1 miss penalty* is the number of cycles the CPU waits when there is an L1 cache miss, and *L1 miss rate* is the number of L1 misses per memory reference.

From these equations we obtain:

$$\begin{aligned} \text{execution time} &= (\text{CPI} \times \text{IC} + \\ &\quad \text{IC} \times \text{L1 misses per instruction} \\ &\quad \times \text{L1 miss penalty}) \\ &\quad \times \text{clock cycle time} \end{aligned} \quad (8)$$

We also see that

$$\begin{aligned} \text{L1 miss penalty} &= \text{L2 hit time} + \text{L2 miss rate} \times \\ &\quad \text{L2 miss penalty} \end{aligned} \quad (9)$$

where *L2 hit time* is the number of cycles to load an L1 cache line from L2 cache and *L2 miss penalty* = *memory hit time* is the number of cycles needed to load an L2 cache line from main memory.

We use Equations 8 and 9 to estimate the run time of Floyd's algorithm. Since the L2 hit time and L2 miss penalty are architecture dependent and not available to us, we use typical numbers for these—the L2 hit time is assumed to be between 6 and 10 cycles and the L2 miss penalty is assumed to be 50 cycles. For the L1 misses per instruction and the L2 miss rate we use data obtained by using the cache simulator Shade on Floyd's algorithm. Table 11 gives this data.

**Table 1.** Cache simulator data for algorithm of Figure 1

Matrix size	L1 misses per instruction (%)	L2 miss rate (%)
480	3.950	18.42
800	4.106	19.17
1600	4.133	19.42
2400	4.826	19.64
3200	5.553	20.07

Now we obtain a lower bound on the run time of a cache optimized version of Floyd's algorithm. Substituting Equation 7 into Equation 8 and making the reasonable assumption that cache optimization will not decrease the total number of memory references (i.e., the number of memory references for the cache optimized code is at least  $IC * \text{memory references per instruction}$  where  $IC$  and  $\text{memory references per instruction}$  are for `AllPairs`) yields

$$\begin{aligned} \text{execution time} \geq & (CPI \times IC + \\ & IC \times \text{memory references per} \\ & \text{instruction} \\ & \times L1 \text{ miss rate} \times \\ & L1 \text{ miss penalty}) \\ & \times \text{clock cycle time} \end{aligned} \quad (10)$$

The cache simulator gives 0.35 as the memory references per instruction for `AllPairs`. Substituting 0.35 for the number of memory references per instruction and the right side of Equation 9 for the L1 miss penalty into Equation 10 we get

$$\begin{aligned} \text{execution time} \geq & (CPI \times IC + 0.35 \times IC \times \\ & L1 \text{ miss rate} \times (L2 \text{ hit time} + \\ & L2 \text{ miss rate} \times L2 \text{ miss penalty})) \\ & \times \text{clock cycle time} \end{aligned} \quad (11)$$

We may obtain a lower bound for the L1 and L2 miss rate by determining the minimum number of L1 and L2 misses that every reorganized version of Figure 1 must make. Since we intend to declare `i`, `j`, `k`, and `n` as register variables [8], references to these variables do not access cache and so do not cause any cache misses. Therefore, we focus on cache misses attributable to the array `A`. For our analysis we use the cache characteristics of the Sun Enterprise 4000/5000 that are shown in Table 2. By direct mapped we mean that each byte of main memory has exactly one byte of cache to which it may be mapped. The line size of a cache

**Table 2.** Cache characteristics of the Sun Enterprise 4000/5000

Cache	Associativity	Cache size	Line size
L1	Direct mapped	16KB	32 bytes
L2	Direct mapped	4MB	64 bytes

gives the unit of memory transfer. So in the Sun Enterprise 4000/5000 an L1 cache miss results in a 32-byte block of data being transferred from L2 cache into L1 cache. The transferred block is one-half of an L2 line.

For the analysis we assume that  $A$  is an integer array and that each integer is 4 bytes. Since Floyd's algorithm accesses each of the  $n^2$  elements of  $A$ , all  $n^2$  elements of  $A$  must get to L1 cache at some time. Each L1 cache miss brings in exactly 32 bytes of data (i.e., 8 elements of  $A$ ). Therefore, the number of L1 cache misses is at least  $n^2/8$ . By a similar reasoning, the number of L2 cache misses is at least  $n^2/16$ . Further, Floyd's algorithm makes  $3n^3$  read accesses to  $A$  (i.e., in the right side of the `min` statement of Figure 1) and  $n^3$  write accesses (the left side of the `min` statement). We note that when the `min` statement of Figure 1 is coded as an `if` statement, write accesses are made only when the new  $a[i][j]$  value is smaller than the old one. In this case the number of write accesses ranges from 0 to  $n^3$ . To keep the analysis simple, we use  $n^3$  as the write access count. The total number of accesses to  $A$  (read and write) is  $4n^3$ . Therefore,

$$\begin{aligned} \text{L1 miss rate} &= \text{L1 misses per } A \text{ reference} \\ &\geq n^2/8/(4n^3) = 1/(32n) \end{aligned} \quad (12)$$

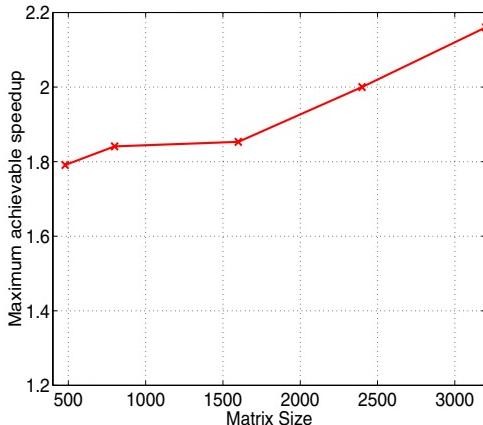
$$\begin{aligned} \text{L2 miss rate} &= \text{L2 misses per } A \text{ reference} \\ &\geq n^2/16/(4n^3) = 1/(64n) \end{aligned} \quad (13)$$

The equality between the miss rate and the misses per  $A$  reference follows from our assumption that variables other than  $A$  will be register variables and so all memory references are to elements of  $A$ . Since we assume that cache optimization does not reduce the number of  $A$  references, these bounds apply to all cache optimized versions of AllPairs.

Substituting the bounds of Equations 12 and 13 into Equation 11, we get the following lower bound on the run time of a cache optimized version of Floyd's algorithm.

$$\begin{aligned} \text{execution time} &\geq (\text{CPI} \times IC + \\ &\quad 0.35 \times IC \times 1/(32n) \times \\ &\quad (L2 \text{ hit time} + \\ &\quad 1/(64n) \times L2 \text{ miss penalty}) \\ &\quad \times \text{clock cycle time} \end{aligned} \quad (14)$$

Dividing Equation 8 by Equation 14 yields an upper bound on the speedup obtainable by optimizing cache utilization. Figure 2 plots this upper bound when  $CPI$  ranges between 1 and 2, L2 hit time ranges from 6 to 10 cycles, and L2 miss penalty is 50 cycles. The L1 misses per instruction and the L2 miss rate are taken from Table 1. Figure 2 gives the maximum speedup we can get by optimizing the cache usage of Floyd's algorithm on typical computers that have a two-level cache.



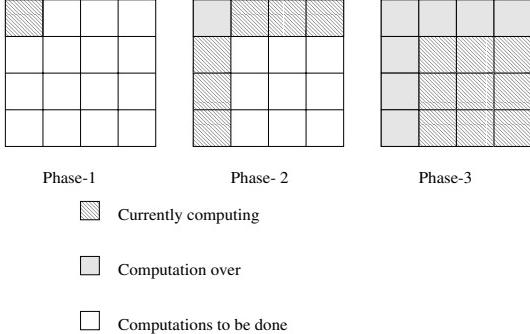
**Fig. 2.** Maximum achievable speedup for different matrix sizes

## 4 Blocked Version of Floyd's Algorithm

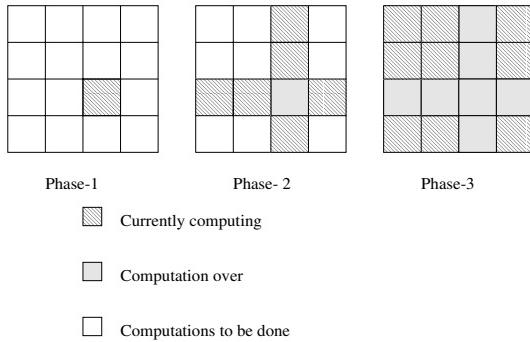
### 4.1 The Algorithm

We partition the cost adjacency matrix into submatrices of size  $B \times B$ .  $B$  is called the *blocking factor*. Although this is not necessary, we assume, for simplicity, that  $B$  divides  $n$ . Our blocked version of Floyd's algorithm (Figure 11) will perform  $B$  iterations of the outermost loop of Figure 11 on each  $B \times B$  block of  $A$  before advancing to the next  $B$  iterations. It is convenient to think of each set of  $B$  iterations as divided into three phases. (Note that our implementation does not actually preform the computation in the three phase order described below.) For example, in phase 1 of the first set of  $B$  iterations, Equation 2 is used to compute  $D^k = A^k$ ,  $1 \leq k \leq B$  for the elements in the top left block, block (1,1). Since these  $B$  iterations access only the  $A$  elements within block (1,1), we say that block (1,1) is a self-dependent block in the first  $B$  iterations.

In phase 2 of the first  $B$  iterations a modified Equation 2 is used to compute  $D^k$ ,  $1 \leq k \leq B$  for the remaining blocks (1,\*) and (\*,1) that are on the same



(a) Phases when (1,1) is the self-dependent block

(b) Phases when block  $(t, t)$  is the self-dependent block**Fig. 3.** Blocks computed in each phase

row or column as the self-dependent block. For the remaining  $(1,*)$  blocks the modified Equation 2 is

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^B(i, k) + D^{k-1}(k, j)\}, k \geq 1 \quad (15)$$

where  $D^0(i, j) = A^0(i, j)$ . For the remaining  $(*,1)$  blocks the modified Equation 2 is

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{k-1}(i, k) + D^B(k, j)\}, k \geq 1 \quad (16)$$

In phase 3  $D^k$ ,  $1 \leq k \leq B$  is computed for the remaining blocks (i.e., for blocks that are not on the same row or column as the self-dependent block). This computation is done using Equation 17

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^B(i, k) + D^B(k, j)\}, k \geq 1 \quad (17)$$

Phase 3 is followed by the next round of  $B$  iterations. These are also done in three phases. This time block (2,2) is the self-dependent block.  $D^k$ ,  $B < k \leq 2B$  are computed for the self-dependent block in phase 1 using the equation

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{k-1}(i, k) + D^{k-1}(k, j)\} \quad (18)$$

In phase 2  $D^k$ ,  $B < k \leq 2B$  are computed for the remaining blocks that are on the same row or column as the self-dependent block and in phase 3  $D^k$ ,  $B < k \leq 2B$  is computed for the blocks that are not on the same row or column as the self-dependent block. The phase 2 computation uses the following equation for the (2,\*) blocks

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{2B}(i, k) + D^{k-1}(k, j)\} \quad (19)$$

The (\*,2) blocks use the following equation

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{k-1}(i, k) + D^{2B}(k, j)\} \quad (20)$$

and the phase 3 blocks use the equation

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{2B}(i, k) + D^{2B}(k, j)\} \quad (21)$$

The following equations are used to compute the  $(t, *)$ ,  $(*, t)$ , and phase 3 blocks, respectively.

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{tB}(i, k) + D^{k-1}(k, j)\} \quad (22)$$

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{k-1}(i, k) + D^{tB}(k, j)\} \quad (23)$$

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{tB}(i, k) + D^{tB}(k, j)\} \quad (24)$$

## 4.2 Correctness of Blocked Algorithm

The  $D^k(i, j)$  values computed by the blocked algorithm are not necessarily the same as the  $A^k(i, j)$  values computed by the unblocked algorithm. For example, when  $B = 4$ , the unblocked algorithm computes  $A^1(4, 7) = \min\{A^0(4, 7), A^0(4, 1) + A^0(1, 7)\}$ , whereas the blocked algorithm computes  $D^1(4, 7) = \min\{D^0(4, 7), D^4(4, 1) + D^0(1, 7)\} = \min\{A^0(4, 7), D^4(4, 1) + A^0(1, 7)\}$ . Since  $D^4(4, 1) = A^4(4, 1)$  is  $\leq A^0(4, 1)$ ,  $D^1(4, 7) \leq A^1(4, 7)$ .

To establish the correctness of the blocked algorithm we must show that  $D^n(i, j) = A^n(i, j)$  for all  $i$  and  $j$ . That is, even though  $D^k(i, j)$  and  $A^k(i, j)$  may not be equal for  $k < n$ , the values agree in the end when  $k = n$ . Actually we will show that  $A$  and  $D$  agree at the end of each set of  $B$  iterations. That is,  $D^k(i, j) = A^k(i, j)$  for all  $i$  and  $j$  whenever  $k$  is a multiple of  $B$ . Hence  $D^n(i, j) = A^n(i, j)$  for all  $i$  and  $j$ .

Let  $k = qB$ . The proof is by induction on  $q$ . We may show that  $D^k(i, j) = A^k(i, j)$  for all  $i$  and  $j$  for  $0 \leq q \leq n/B$ . The proof is omitted from this version of the paper.

### 4.3 Optimal Blocking Factor

When computing the  $D$  values in a block during any round (i.e., an iteration of the outermost loop) of function `BoundedAllPairs`, at most three blocks are active. The computation for the self-dependent block accesses elements only in the self-dependent block. So during the self-dependent block computation only 1 block is active. The computation for a block  $R$  that is on the same row or column as the self-dependent block accesses elements in  $R$  as well as elements in the self-dependent block. Therefore, 2 blocks are active during the computation for  $R$ . For a block  $R$  that is not on the same row or column as the self-dependent block, `BlockedAllPairs` accesses elements from 3 blocks—block  $R$ , the block that is in the same row as the self-dependent block and the same column as  $R$ , and the block that is in the same column as the self-dependent block and in the same row as  $R$ . Therefore, L1 cache misses are minimized by choosing the largest block size  $B$  such that 3 block loads of the array  $D$  fit into L1 cache. Suppose that the elements of  $D$  are 4-byte integers and that our L1 cache capacity is  $C$  bytes and that each L1 cache line is  $S$  bytes. We must choose  $B$  to be the largest integer such that  $3B^2 * 4 \leq C$  (equivalently,  $B \leq \sqrt{C/12}$ ) and  $B$  is a multiple of  $S/4$ . The second requirement is necessary as the smallest unit of data brought into L1 cache is  $S$  bytes and these  $S$  bytes are contiguous bytes of memory.

For the Sun Ultra Enterprise 4000/5000  $C = 16K$  and  $S = 32$ . Therefore, the blocking factor should be the largest integer that is  $\leq \sqrt{C/12} = 37$  and is a multiple of  $32/4 = 8$ . That is, we should use  $B = 32$  as the blocking factor. For the SGI O2  $C = 32K$  and  $S = 32$ . The optimal blocking factor for the SGI O2 is the largest integer that is  $\leq \sqrt{C/12} = 52$  and is a multiple of  $32/4 = 8$ . This optimal blocking factor is 48.

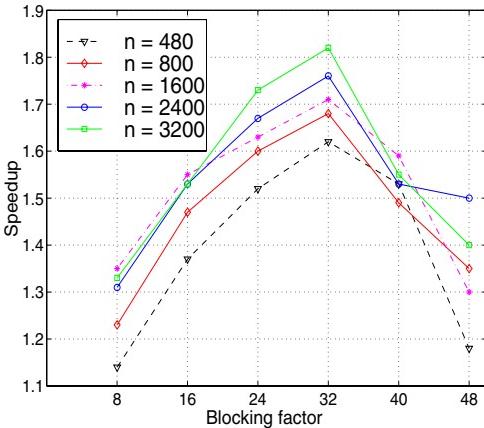
## 5 Experimental Results

The speedup of our blocked shortest paths algorithm relative to the standard unblocked algorithm was measured by programming the two algorithms in C++ (the g++ compiler with optimization option o5 was used) and running the two programs on a Sun Ultra Enterprise 4000/5000 and an SGI O2. Both programs were compiled using the highest-level of compiler optimization possible.

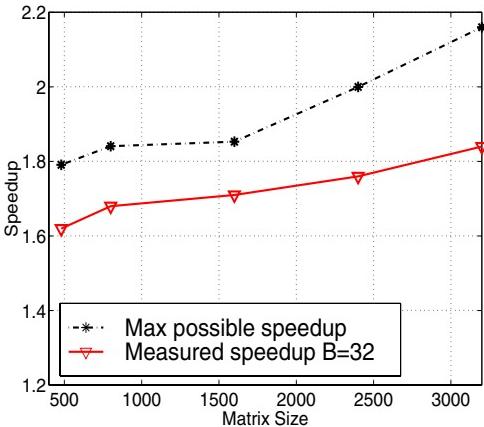
We first present the results for the SUN Ultra Enterprise. Figure 4 gives the measured speedups for different blocking factors and different  $n$ . As predicted by our analysis, the optimal blocking factor is 32 for all  $n$ .

Figure 5 compares the speedup obtained by `BlockedAllPairs` and the maximum speedup possible by optimizing cache utilization. The curve for maximum possible speedup is that of Figure 2.

The speedup obtained by `BlockedAllPairs` is fairly close to the maximum possible. One reason we do not achieve the predicted maximum speedup is that the total instruction count for `BlockedAllPairs` is more than that for `AllPairs`. Recall that in determining the maximum speedup curve of Figure 2 we assumed that the instruction count for the cache optimized algorithm is the same as that of `AllPairs`.



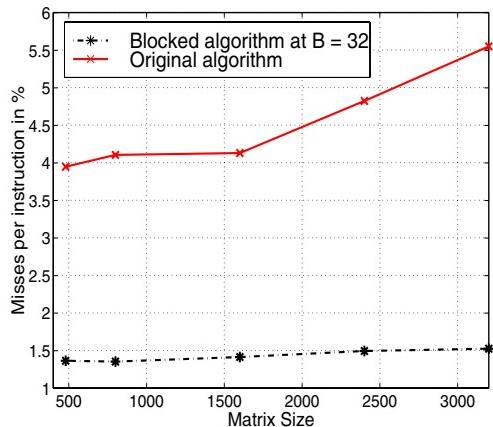
**Fig. 4.** Speedup of `BlockedAllPairs` on a Sun Ultra Enterprise



**Fig. 5.** Measured and maximum possible speedup

Figure 6 gives the *L1 misses per instruction* for the unblocked and blocked versions of Floyd's algorithm. The data for this figure were obtained using the cache simulator Shade. As expected the blocked code shows better cache utilization.

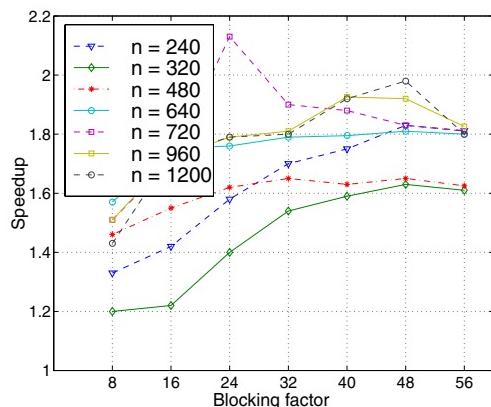
Table 3 shows the cache details for the SGI O2 computer and Figure 7 shows the speedup obtained by the blocked algorithm on an SGI O2. Except for one anomaly, maximum speedup is obtained when the blocking factor is the predicted optimal factor of 48.



**Fig. 6.** Misses per instruction for unblocked and blocked algorithms

**Table 3.** Cache configuration of SGI

Cache type	Cache size
L1	32KB
L2	1MB



**Fig. 7.** Speedup obtained by `BlockedAllPairs` on an SGI O2

## 6 Conclusion

We have developed a blocked version of Floyd's all-pairs shortest-paths algorithm. Experimental results show that the blocked version obtains speedups close to the maximum possible for a cache optimized version of Floyd's algorithm.

## References

1. W. AbuSufah, D. J. Kuck, and D. H. Lawrie. Automatic program transformation for virtual memory computers. In *Proc. of the 1979 National Computer Conference*, pages 969–974, New York, 1979.
2. A. Aggarwal, K. Chandra, and M. Snir. A model for hierarchical memory. In *The 19th Annual ACM Symposium on Theory of Computing*, pages 305–314, New York, 1987.
3. A. Aggarwal, K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *The 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 204–216, Los Angeles, CA, 1987.
4. A. Aggarwal, M. Horowitz, and Hennessey. An analytical cache model. *The ACM Transactions on Computer Systems*, 7(2):184–215, 1989.
5. I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. *Proc. 12th International Parallel Processing Symposium '98. (IPPS'98), Orlando, Florida*.
6. I. Al-Furaih and S. Ranka, Ibraheem Al-Furaih and Sanjay Ranka, “Practical Algorithms for Internal and External Sorting”, *Proc. the Second International Conference on Parallel and Distributed Computing and Networks (PDCN'98), Brisbane, Australia, 14-16 December 1998*.
7. B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3):72–109, 1994.
8. D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *In Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, New York, 1990.
9. D. Gannon and W. Jalby. The influence of memory hierarchy on algorithm organization: Programming FFTs on a vector multiprocessor. In *The Characteristics of Parallel Algorithms*, MIT Press, Cambridge, 1987.
10. G. H. Golub and C. F. Van Loan. Matrix Computations. *Johns Hopkins University Press, Baltimore*, 1989.
11. E. Horowitz, S. Sahni, and S. Rajasekaran. *Computer Algorithms*. Computer Science Press, New York, 1998.
12. M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *ACM*, 26:63–74, 1991.
13. A. LaMarca and R. E. Ladner. The influences of caches on the performance of heaps. *The ACM Journal of Experimental Algorithms*, 1(4), 1996.
14. A. LaMarca and R. E. Ladner. The influences of caches on the performance of sorting. In *The ACM-SIAM Symposium on Discrete Algorithms*, pages 370–379, New Orleans, Louisiana, 5-7 January, 1997.
15. M. Martonosi, A. Gupta, and T. Anderson. Memspy: analyzing memory system bottlenecks in programs. In *Proceedings of the 1992 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, Newport, Rhode Island, 1992.

16. A. C. McKeller and E. G. Coffman. The organization of matrices and matrix operations in a paged multiprogramming environment. *CACM*, 12(3):153–165, 1969.
17. Sun Microsystems. Introduction to Shade. Manual, Sun Microsystems, Mountain View, CA, 1998.
18. D. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Analysis*. Morgan Kaufmann, San Mateo, CA, 1996.
19. J. P. Singh, H. S. Stone, and D. F. Thiebaut. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Transactions on Computers*, 41(7):811–825, 1992.
20. Larry Stewart. Programming to Optimize Cache Memory on the SUN Ultrasparc-II Processor. Master's thesis, University of Florida, Gainesville, FL, April 1999.
21. P. Sulyatycze and K. Ghose. Caching-efficient multithreaded fast multiplication of sparse matrices. *Proceedings 12the International Parallel Processing Symposium*, 117–123, 1998.
22. O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the 1994 ACM SIGMETRICS: Conference on Measurement and Modelling of Computer Systems*, pages 261–271, Nashville, Tennessee, 1994.
23. O. Temam, C. Fricker, and W. Jalby. Influence of cross-interference on blocked loops: A case study with matrix-vector multiply . *The ACM Transactions on Programming Languages and Systems*, 17(4):561–575, 1994.
24. H. Wen and J. L. Baer. Efficient trace driven simulation methods for cache performance analysis. *The ACM Transactions on Computer Systems*, 9(3):222–241, 1991.
25. M. E. Wolf and M. S. Lam. A data locality optimizing. In *In Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Ontario, Canada, 1991.

# On External-Memory MST, SSSP, and Multi-way Planar Graph Separation

(Extended Abstract)

Lars Arge<sup>1,\*</sup>, Gerth Stølting Brodal<sup>2,\*\*</sup>, and Laura Toma<sup>1,\*\*\*</sup>

<sup>1</sup> Duke University, Durham, NC 27708–0129, USA

<sup>2</sup> University of Aarhus, DK-8000 Århus C, Denmark

**Abstract.** Recently external memory graph algorithms have received considerable attention because massive graphs arise naturally in many applications involving massive data sets. Even though a large number of I/O-efficient graph algorithms have been developed, a number of fundamental problems still remain open. In this paper we develop an improved algorithm for the problem of computing a minimum spanning tree of a general graph, as well as new algorithms for the single source shortest paths and the multi-way graph separation problems on planar graphs.

## 1 Introduction

Recently external memory graph algorithms have received considerable attention because massive graphs arise naturally in many applications involving massive data sets. One example of a massive graph is AT&T’s 20TB phone-call data graph [11]. Other examples of massive graphs arise in Geographic Information Systems (GIS). For instance, GIS terrains are often represented using planar graphs and many common GIS problems can be formulated as standard graph problems (Arc/Info [4], the most commonly used GIS package, contains functions that correspond to computing depth-first, breadth-first, and minimum spanning trees, as well as shortest paths and connected components). When working with such massive graphs the I/O-communication, and not the internal memory computation time, is often the bottleneck. Designing efficient external memory algorithms for such problems can thus lead to considerable runtime improvements, as for example illustrated in our previous work [7].

Even though a large number of I/O-efficient graph algorithms have been developed in recent years, a number of important problems still remain open. For

\* Supported in part by the National Science Foundation through ESS grant EIA-9870734, RI grant EIA-9972879 and CAREER grant EIA-9984099. E-mail: [large@cs.duke.edu](mailto:large@cs.duke.edu).

\*\* BRICS (Basic Research in Computer Science, Center of Danish National Research Foundation). Supported in part by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT). E-mail: [gerth@brics.dk](mailto:gerth@brics.dk).

\*\*\* Supported in part by the National Science Foundation through ESS grant EIA-9870734 and RI grant EIA-9972879. E-mail: [laura@cs.duke.edu](mailto:laura@cs.duke.edu).

example, developing efficient algorithms for basic problems such as breadth-first search and depth-first search remain open. In this paper we develop I/O-efficient algorithms for the minimum spanning tree (MST) and single source shortest paths (SSSP) problems, as well as for multi-way planar graph separation.

### 1.1 Problem Statement

MST and SSSP are well-known problems on a weighted graph  $G = (V, E)$ : MST is the problem of finding a spanning tree for  $G$  of minimum weight and SSSP is the problem of finding the shortest paths from a given source vertex in  $G$  to all other vertices in  $G$  (the length of a path is the sum of the weights of the edges on the path).

Consider an undirected graph  $G = (V, E)$ <sup>1</sup>. An  $f(V)$ -separator of  $G$  is a subset  $S$  of the vertices of  $G$  of size  $f(V)$  such that the removal of  $S$  disconnects  $G$  into two subgraphs  $G_1$  and  $G_2$ , each of size at most  $\frac{2V}{3}$ . Lipton and Tarjan [23] proved that any planar graph has an  $O(\sqrt{V})$ -separator and gave a linear time algorithm for finding such a separator. Using this result recursively, a planar graph can be decomposed into  $\Theta(\frac{V}{R})$  subgraphs  $G_i$  with  $O(R)$  vertices each and  $O(\frac{V}{\sqrt{R}})$  separator vertices, such that there is no edge between a vertex in  $G_i$  and a vertex in  $G_j$  for  $i \neq j$ . We call such a decomposition a *multi-way planar graph separation* of  $G$ . Graph separation is often used in the design of divide-and-conquer algorithms.

Throughout this paper we assume that the input graph  $G$  is given in edge-list representation. If  $G$  is planar we assume it is embedded in the plane. We also assume without loss of generality that  $G$  is connected and that no two edges have the same weight. In some of our algorithms we will assume that a breadth-first-search tree  $T$  of  $G$  is given. In such cases we assume that  $T$  is represented implicitly by storing with each vertex  $u$  in  $G$  its parent in  $T$  and marking every edge of  $G$  as either a tree or a non-tree edge.

### 1.2 Previous Results on I/O-Efficient Graph Algorithms

We work in the standard two-level I/O model with one (logical) disk [3,20]. The model defines the following parameters:

$$N = V + E,$$

$M$  = number of vertices/edges that can fit into internal memory,

$B$  = number of vertices/edges per disk block,

where  $M < N$  and  $1 \leq B \leq M^{1/(2+\varepsilon)}$ , for some  $\varepsilon > 0$ .<sup>2</sup> An *Input/Output* (or simply *I/O*) involves reading (or writing) a block from disk into (from) internal

<sup>1</sup> For convenience we will use the name of a set to denote both the actual set and its cardinality.

<sup>2</sup> Often it is only assumed that  $B \leq M/2$  but sometimes, as in this paper, the very realistic assumption that the main memory is capable of holding  $B^2$  elements is made (or as here,  $B^{2+\varepsilon}$  for some  $\varepsilon > 0$ ).

memory. Our measure of performance of an algorithm is the number of I/Os it performs. The number of I/Os needed to read  $N$  contiguous items from disk is  $\text{scan}(N) = \Theta(\frac{N}{B})$  (the scanning bound), and the number of I/Os required to sort  $N$  items is  $\text{sort}(N) = \Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$  [3] (the sorting bound). In practice the difference between an algorithm doing  $N$  I/Os and one doing  $\text{scan}(N)$  or  $\text{sort}(N)$  I/Os can be significant [7].

**Table 1.** Best known upper bounds for basic graph theoretic problems.

Problem	General undirected graphs	
DFS	$O\left(\frac{V}{M} \cdot \frac{E}{B} + V\right)$	[12]
	$O\left((V + \text{scan}(E)) \cdot \log \frac{V}{B} + \text{sort}(E)\right)$	[22]
BFS	$O(V + \frac{E}{V} \cdot \text{sort}(V))$	[25]
CC	$O\left(\text{sort}(E) \cdot \log \log \frac{VB}{E}\right)$	[25]
MST	$O\left(\text{sort}(E) \cdot \log \frac{V}{M}\right)$	[12]
	$O\left(\text{sort}(E) \cdot \log B + \text{scan}(E) \cdot \log V\right)$	[22]
SSSP	$O\left(V + \frac{E}{B} \cdot \log \frac{V}{B}\right)$	[22]

I/O-efficient graph algorithms have been considered by a number of authors [12, 5, 6, 10, 12, 16, 19, 22, 24, 25, 26, 29]. Table 1 reviews the best known algorithms for basic graph theoretic problems on general undirected graphs. For directed graphs the best known algorithm for breadth-first search (BFS) and depth-first search (DFS) use  $O\left((V + \text{scan}(E)) \cdot \log \frac{V}{B} + \text{sort}(E)\right)$  I/Os [10]. Lower bound results were proved in [6, 12, 25]. Note that no  $O(\text{sort}(E))$  (deterministic) algorithm is known for *any* of the problems, and that the best known algorithms for DFS, BFS and SSSP require  $\Omega(V)$  I/Os. MST and connected components (CC) can be solved in  $O(\text{sort}(E))$  I/Os with randomized algorithms [12, 1].

Improved algorithms have been developed for several special classes of graphs. For trees,  $O(\text{sort}(N))$  algorithms are known for BFS and DFS numbering, Euler tour computation, expression tree evaluation, topological sorting, as well as several other problems [10, 12]. For planar graphs,  $O(\text{sort}(N))$  algorithms are known for CC and MST [12]. For grid graphs  $O(\text{sort}(N))$  algorithms are known for BFS and SSSP, and an  $O(\text{scan}(N))$  algorithm for CC [7]. See [30] for a complete reference.

Given that even very basic graph problems seem hard to externalize, it is natural to try to reduce the problems to one another. A first step in this direction was taken by Hutchinson *et al.* [19] who considered the problem of computing an  $O(\sqrt{N})$ -separator of a planar graph I/O-efficiently. Given a BFS tree they showed how to compute a separator in  $O(\text{sort}(N))$  I/Os. Given this algorithm, it is straightforward to solve the multi-way planar graph separation problem in  $O(\log \frac{N}{R} \cdot \text{sort}(N))$  I/Os, simply by applying the algorithm recursively.

### 1.3 Our Results

In Section 2, we give an  $O(\text{sort}(E) \cdot \log \log \frac{V^B}{E}) = O(\text{sort}(E) \cdot \log \log B)$  algorithm for the MST problem on general undirected weighted graphs, improving the previous bound of  $O(\text{sort}(E) \cdot \log B + \text{scan}(E) \cdot \log V)$  [22]. The algorithm uses the same general idea as the CC algorithm by Munagala and Ranade [25] and consists of two phases: first a vertex contraction algorithm is used to reduce the number of vertices to  $O(\frac{E}{B})$ , and then an  $O(V + \text{sort}(E))$  MST algorithm is used on the reduced graph. The new contraction algorithm uses ideas similar to the ones used in [8, 14, 25], as well as a simplified version of the basic contraction step used in previous MST algorithms [8, 12, 13, 14, 22, 25, 28]. The new  $O(V + \text{sort}(E))$  MST algorithm is a modified version of Prim's algorithm. It remains a challenging open problem to develop an  $O(\text{sort}(E))$  MST algorithm.

In Section 3 and 4, we show that the multi-way planar graph separation problem and the SSSP problem can be reduced to the BFS problem in  $O(\text{sort}(N))$  I/Os: In Section 3, we give an  $O(\text{sort}(N))$  algorithm for the multi-way planar graph separation problem given a BFS tree. The algorithm improves the straightforward bound of  $O(\log \frac{N}{R} \cdot \text{sort}(N))$  I/Os and uses a divide-and-conquer algorithm based on ideas from [18]. In Section 4 we show how to use this result to solve the SSSP problem in  $O(\text{sort}(N))$  I/Os. The algorithm is a generalization of our SSSP algorithm on grid graphs [7] and uses ideas similar to the ones utilized by Frederickson [17]. We believe that our  $O(\text{sort}(N))$  graph separation algorithm might prove helpful in reducing other problems on planar graphs to the BFS problem. It remains a challenging problem to develop an  $O(\text{sort}(E))$  BFS algorithm. Another interesting open problem is if it is possible to develop an  $O(\text{sort}(E))$  BFS algorithm for a planar graph given a multi-way separation of the graph.

## 2 Minimum Spanning Tree on General Graphs

In this section we describe our MST algorithm on general undirected weighted graphs. The basic idea is to reduce the number of vertices to  $\frac{E}{B}$  using an  $O(\text{sort}(E))$  vertex reduction algorithm  $O(\log \log \frac{V^B}{E})$  times, and then use an  $O(V + \text{sort}(E))$  MST algorithm on the resulting graph. The overall I/O complexity will thus be  $O(\text{sort}(E) \cdot \log \log \frac{V^B}{E} + \frac{E}{B} + \text{sort}(E)) = O(\text{sort}(E) \cdot \log \log \frac{V^B}{E})$  I/Os. In Section 2.1 we first describe the  $O(V + \text{sort}(E))$  MST algorithm, and in Section 2.2 we then describe the reduction algorithm. The MST result is summarized in the following theorem.

**Theorem 1.** *The MST of an undirected weighted graph can be found in  $O(\text{sort}(E) \cdot \log \log \frac{V^B}{E})$  I/Os.*

### 2.1 An $O(V + \text{sort}(E))$ MST Algorithm

Our algorithm is a modified version of Prim's internal memory algorithm [15]. The idea of Prim's algorithm is to grow the MST iteratively from a source node

while maintaining a priority queue on the vertices not included in the MST so far; the priority of a vertex is the weight of the minimum edge connecting it to the current MST. The algorithm repeatedly extracts the minimum priority vertex  $v$ , adds it to the MST, and updates the priority of the vertices  $u$  adjacent to  $v$ . Specifically, the weight  $w$  of edge  $(v, u)$  is compared with the priority of vertex  $u$  in the priority queue, and an update is performed if  $w$  is smaller than the current priority. Prim's algorithm cannot be implemented efficiently in external memory, the main reason being that the current priority of a given vertex cannot in general be obtained without doing one I/O. A direct implementation would thus lead to an  $O(E)$  I/O bound. Previously known algorithms [12, 22] rely instead on vertex contraction methods [8, 13, 14].

Our modification of Prim's algorithm consists of storing edges in the priority queue instead of vertices. During the algorithm the priority queue contains (at least) all edges connecting vertices in the current MST with vertices not in the tree. The queue can also contain edges between two vertices in the MST. The algorithm works as follows: Repeatedly perform *extract\_min* to extract the minimum weight edge  $(u, v)$  from the priority queue. If  $v$  is already in the MST the edge is discarded. Otherwise  $v$  is included in the MST and all edges incident to  $v$ , except  $(v, u)$ , are inserted in the priority queue. The key to the I/O-efficiency of the algorithm is that because we store edges in the priority queue we have a simple way of checking whether a vertex is already included in MST — as all edges incident to  $v$  are inserted in the priority queue when  $v$  is included in the MST, it follows that if both  $u$  and  $v$  are in the MST when processing an edge  $e = (u, v)$ , the edge  $e$  must appear in the priority queue twice. Thus we can check if  $v$  is already included in the MST simply by performing one more *extract\_min* and checking if it returns the same edge  $e$  (recall that we assume that no two edges have the same weight).

The algorithm performs at least one I/O for each vertex which is included in the MST in order to read its adjacent vertices (traverse its adjacency lists). Thus processing all vertices and edges takes  $V + \frac{E}{B}$  I/Os. It also performs  $O(E)$  *insert*'s and *extract\_min*'s on the priority queue. Using an external priority queue [5, 9] supporting these operations in  $O(\frac{1}{B} \log_{M/B} \frac{N}{B})$  I/Os amortized we obtain:

**Lemma 1.** *The MST of an undirected weighted graph can be computed in  $O(V + \text{sort}(E))$  I/Os.*

## 2.2 MST Vertex-Reduction Algorithm

Our MST vertex reduction algorithm is obtained using ideas from the connected-component algorithm of Munagala and Ranade [25] and the notion of “blocking values”. The standard MST algorithm based on vertex contraction proceeds in  $\lceil \log V \rceil$  phases [12, 22]. In each phase the minimum cost edge adjacent to every vertex  $v$  is selected and output as part of the MST and the vertices connected by the selected edges are contracted to supervertices. Let the size of a supervertex be the number of vertices it contains from the original graph. After the  $i$ th phase the size of every supervertex is at least  $2^i$ . Since one contraction phase can be

performed in  $O(\text{sort}(E))$  I/Os [12] this results in an  $O(\text{sort}(E) \cdot \log V)$  algorithm. The algorithm in [22] utilizes that a contraction step can be performed more efficiently after  $O(\log B)$  phases and obtains an  $O(\text{sort}(E) \cdot \log B + \text{scan}(E) \cdot \log V)$  algorithm.

Our algorithm runs for  $\lceil \log \frac{VB}{E} \rceil$  phases after which the number of supervertices is at most  $\frac{E}{B}$ . Furthermore we reduce the number of I/Os used in the process by dividing the  $\lceil \log \frac{VB}{E} \rceil$  phases into *superphases* requiring  $O(\text{sort}(E))$  I/Os each: Let  $N_i = 2^{(3/2)^i}$ , i.e.  $N_{i+1} = N_i \sqrt{N_i}$ . Superphase  $i$ , for  $i \geq 0$ , consists of  $\lceil \log \sqrt{N_i} \rceil$  phases. In a preprocessing step we run the basic vertex contraction algorithm once to insure that the number of vertices before superphase 0 is  $V_0 \leq \frac{V}{N_0} = \frac{V}{2}$ . We will maintain the invariant that before superphase  $i$  the number of supervertices is at most  $\frac{V}{N_i}$ . To reduce the number of vertices to at most  $\frac{E}{B}$  it is therefore sufficient to perform  $3 + \lceil \log_{3/2} \lceil \log \frac{VB}{E} \rceil \rceil$  superphases and we obtain the  $O(\text{sort}(E) \cdot \log \log \frac{VB}{E})$  algorithm.

The phases in each superphase only work on a subset of the (remaining) edges. The edge subsets are chosen in order to allow each supervertex to grow by a factor of  $\sqrt{N_i}$  in superphase  $i$ . Let  $G_i = (V_i, E_i)$  be the graph just prior to superphase  $i$ . We construct a graph  $G'_i = (V_i, E'_i)$ , where  $E'_i$  is a subset of  $E_i$ . For each vertex  $v$ ,  $E'_i$  contains the  $\lceil \sqrt{N_i} \rceil$  lightest edges adjacent to  $v$ . Heavier edges  $e = (v, u)$  adjacent to  $v$  are only included in  $E'_i$  if  $e$  is among the  $\lceil \sqrt{N_i} \rceil$  lightest edges adjacent to  $u$ . We define the *blocking value* of  $v$  to be the weight of the  $(\lceil \sqrt{N_i} \rceil + 1)$ -th lightest edge adjacent to  $v$ . The set  $E'_i$  and blocking values can be computed using  $O(\text{sort}(E_i))$  I/Os. If we guarantee that  $V_i \leq \frac{V}{N_i}$  as stated above, it follows that  $E'_i \leq 2V_i \lceil \sqrt{N_i} \rceil < 4 \frac{V}{\sqrt{N_i}}$ . As each contraction phase in superphase  $i$  can be performed in  $O(\text{sort}(E'_i))$  I/Os, it follows that superphase  $i$  requires  $O(\text{sort}(E_i) + \text{sort}(E'_i) \cdot \log(\sqrt{N_i})) = O(\text{sort}(E) + \text{sort}(\frac{V}{\sqrt{N_i}}) \cdot \log(\sqrt{N_i})) = O(\text{sort}(E))$  I/Os. After performing all the phases of superphase  $i$  the edges  $E_i - E'_i$ , i.e. the heavy edges which were not included in the sample, need to be re-incorporated in  $E_{i+1}$ . This can be easily be done as in [25] using  $O(\text{sort}(E))$  I/Os in total. Details will appear in the full paper.

The only thing that remains to be described is how the individual phases in superphase  $i$  are performed such that after superphase  $i$  the number of supervertices is at most  $\frac{V}{N_{i+1}}$  and such that only edges that actually belong to the MST are included. A phase is performed as in the basic vertex reduction algorithm: For each vertex  $v$  consider the adjacent edge  $e$  with minimum weight in  $E'_i$ . If the weight of  $e$  is smaller than the blocking value of  $v$ , then we select  $e$  for contraction. If the weight of  $e$  is larger than the blocking value, no edges is selected for  $v$ , since there might be a lighter edge adjacent to  $v$  in  $E_i - E'_i$ . The selected edges are contracted in  $O(\text{sort}(E'_i))$  I/Os (using the algorithm in [12, 22, 25] or a simpler algorithm which we will include in the full version). After the contraction, the blocking value of a supervertex is set to be the minimum of the blocking values of the contracted vertices. The algorithm is correct as a simple induction argument can be used to show that for every supervertex  $v$  the (contracted) edge sample contains all edges adjacent to  $v$  with weight smaller

than the blocking value of  $v$  (i.e. the edges selected in the next phase belong to the MST). If in superphase  $i$  the blocking value of a supervertex  $v$  prevents us from selecting an edge for  $v$  to be included in the MST, then  $v$  must be the contraction of at least  $\sqrt{N_i}$  vertices from  $V_i$ . This follows from the fact that the blocking value of  $v$  corresponds to the blocking value of some vertex  $u$  in  $V_i$  and  $v$  must span the  $\lceil \sqrt{N_i} \rceil$  vertices adjacent to  $u$  in  $E'_i$ . If no blocking value prevents us from selecting an edges for  $v$ , then after  $\lceil \log \sqrt{N_i} \rceil$  phases  $v$  must have size at least  $2^{\log \sqrt{N_i}} = \sqrt{N_i}$ . It follows that superphase  $i$  reduces the number of vertices by a factor of at least  $\sqrt{N_i}$ , i.e. the number of vertices after superphase  $i$  is at most  $\frac{V_i}{\sqrt{N_i}} \leq \frac{V}{N_i \sqrt{N_i}} = \frac{V}{N_{i+1}}$  as claimed by the invariant.

**Lemma 2.** *Let  $G = (V, E)$  be an undirected weighted graph. The MST problem on  $G$  can be reduced to the MST problem on a graph with at most  $\frac{E}{B}$  vertices in  $O(\text{sort}(E) \cdot \log \log \frac{VB}{E})$  I/Os.*

### 3 Multi-way Planar Graph Separation

In this section, we show how to separate a planar graph  $G$  into  $\Theta(\frac{N}{R})$  subgraphs with  $O(R)$  vertices each and a set of  $O(\text{sort}(N))$  separator vertices using  $O(\text{sort}(N))$  I/Os.

Given a BFS tree  $T$  of  $G$ , Hutchinson *et al.* [19] showed how to compute a  $O(\sqrt{N})$ -separator for  $G$  in  $O(\text{sort}(N))$  I/Os. Their algorithm closely follows the algorithm by Lipton and Tarjan [23]: The BFS tree  $T$  has the property that no edge crosses two or more levels, and hence every level in  $T$  is a separator in  $G$ . The basic idea is to use the “middle” level  $\ell_1$  in  $T$  (the level containing the vertex with number  $N/2$  in the BFS numbering) as the separator. Level  $\ell_1$  has the property that the total number of vertices on levels above  $\ell_1$ , as well as in levels below  $\ell_1$ , is less than  $N/2$ . The problem is that  $\ell_1$  might contain more than  $O(\sqrt{N})$  vertices. However, there exists a level  $\ell_0$  above  $\ell_1$  and a level  $\ell_2$  below  $\ell_1$  with  $O(\sqrt{N})$  vertices each, such that  $\ell_2 - \ell_0 \leq \sqrt{N}$  (that is,  $\ell_0$  and  $\ell_2$  are not too far away from  $\ell_1$ ). Levels  $\ell_0$  and  $\ell_2$  divide  $G$  into three subgraphs  $G_0, G_1$  and  $G_2$  consisting of the vertices on the levels above  $\ell_0$ , between  $\ell_0$  and  $\ell_2$  and below  $\ell_2$  respectively, with the property that  $G_0$  and  $G_2$  contain less than  $N/2$  vertices and  $G_1$  has a spanning tree of bounded height  $\sqrt{N}$ . Refer to Fig. 1(a). It is easy to see that in order to find a separator for  $G$  it is enough to find a separator in  $G_1$  [23]. Such a separator can be found using properties of the dual graph of  $G_1$ . The dual graph  $G^* = (V^*, E^*)$  of a planar graph  $G$  is a planar graph with a vertex for each face of  $G$  whose edges are in one-to-one correspondence with the edges of  $G$ . The dual graph  $G^*$  is obtained by placing a vertex in each face of  $G$  and connecting two faces  $f_i$  and  $f_j$  adjacent to a common edge  $e = (u, v)$  of  $G$  with an edge  $(f_i, f_j)$  in  $E^*$ . The edge  $(f_i, f_j)$  in  $G^*$  is called the dual edge of  $(u, v)$  in  $G$ . Let  $E' \subseteq E$  be a subset of edges in  $G$ . It is well known that  $(V, E')$  is a spanning tree of  $G$  if and only if  $(V^*, (E - E')^*)$  is a spanning tree in  $G^*$  [21]. Thus the edges in  $(E - T)^*$  form a spanning tree in  $G^*$  which we denote  $T^\dagger$ . An example is shown in Fig. 2(a). If  $T$  has bounded height  $\sqrt{N}$

then every edge in  $(E - T)$  (and therefore the corresponding edge in  $(E - T)^*$ ) determines a cycle in  $T$  with at most  $2\sqrt{N}$  vertices. Assuming (without loss of generality) that  $G$  is triangulated, Lipton and Tarjan [23] proved that there exists an edge  $e \in (E - T)$  such that the number of vertices inside and outside the cycle defined by  $e$  is  $\leq 2N/3$ , and showed how it can be computed efficiently using a bottom-up traversal of the dual tree  $T^\dagger$ . Hutchinson *et al.* [19] showed how to perform all these operations using  $O(\text{sort}(N))$  I/Os.

As discussed in the introduction, the  $O(\text{sort}(N))$  separator algorithm [19] can be used to develop a recursive  $O(\log \frac{N}{R} \cdot \text{sort}(N))$  multi-way separator algorithm in a straightforward way. The idea in our new  $O(\text{sort}(N))$  algorithm is to obtain  $O(\log_{M/B} \frac{N}{R})$  recursion depth by increasing the fan-out of the separation from 2 to  $\frac{M}{B}$  and implement each step in  $O(\frac{N}{B})$  I/Os. In order to divide the graph in  $\frac{M}{B}$  subgraphs we use ideas similar to the ones used by Goodrich [18]. The general idea is the following: Instead of finding only one level cutting the graph in two halves, we find (roughly)  $\frac{M}{B}$  levels which cut the graph in  $O(\frac{N}{M/B})$ -sized chunks. We then use these levels to find a set of levels with few vertices which divide  $G$  into subgraphs such that each subgraph is either of size  $O(\frac{N}{M/B})$  or has a spanning tree of bounded height  $O(\sqrt{R})$ . We then subdivide the subgraphs with bounded height into graphs of size  $O(R)$  using properties of the dual graph. In Section 3.2 we show how this can be done I/O-efficiently and prove the following lemma:

**Lemma 3.** *A graph  $G$  with a spanning tree  $T$  of height  $H$  can be divided into  $\Theta(\frac{N}{R})$  subgraphs of size  $O(R)$  each and  $O(\frac{N}{R}H)$  separator vertices in total using  $O(\text{sort}(N))$  I/Os.*

After subdividing the bounded height subgraphs we recursively subdivide the subgraphs of size  $O(\frac{N}{M/B})$ . In Section 3.1 we give the details in our algorithm and prove the following:

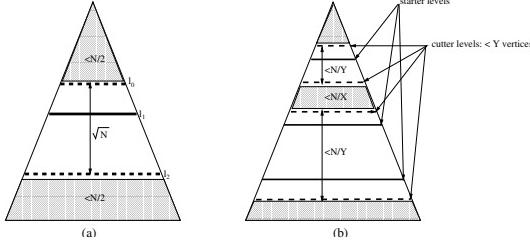
**Theorem 2.** *Let  $G = (V, E)$  be a planar graph and  $T$  a breadth-first search tree for  $G$ . Furthermore assume  $\exists \varepsilon > 0$  such that  $M > B^{2+\varepsilon}$ . For any  $R = \Omega(M)$ ,  $G$  can be partitioned into  $\Theta(\frac{N}{R})$  subgraphs  $G_i$  of size  $O(R)$  each and a set of separator vertices  $S$  of size  $O(\text{sort}(N))$  using  $O(\text{sort}(N))$  I/Os.*

### 3.1 Separating Planar Graphs

In this section we prove Theorem 2 using Lemma 3. Let  $L(i)$  be the total number of vertices on levels 0 through  $i$  of  $T$  and define the *starter levels* to be the levels  $i$  such that the interval  $(L(i), L(i+1)]$  contains a multiple of  $\lceil \frac{N}{X} \rceil$ , for some  $0 < X < N$ . There are at most  $X$  starter levels and the number of vertices between consecutive starter levels is smaller than  $\lceil \frac{N}{X} \rceil$ .

Just like the  $\ell_1$  level in Lipton and Tarjan's algorithm [23], the starter levels divide  $G$  in subgraphs of “small” size. However, as previously, the starter levels can contain too many vertices. Therefore we consider the first level above each starter level, as well as the first level below each starter level containing at most  $Y$  vertices, for some  $0 < Y < N$ . We call these levels the *cutter levels*. The cutter

levels divide  $G$  into  $O(X)$  subgraphs  $G_i$ , consisting of the vertices between two consecutive cutter levels, with the property that if the two cutter levels defining  $G_i$  are within two (consecutive) starter levels then  $G_i$  has size  $O(\frac{N}{X})$ . If the two cutters defining  $G_i$  are not within two consecutive starter levels then  $G_i$  has a spanning tree of depth  $O(\frac{N}{Y})$ . Refer to Fig. 1(b).



**Fig. 1.** (a) Illustration of the planar separator algorithm [23]; (b) Starter and cutter levels in  $T$

subgraphs and  $O(\frac{N_i}{R} \cdot \sqrt{R}) = O(\frac{N}{\sqrt{R}})$  separator vertices using  $O(\text{sort}(N_i))$  I/Os. Note that as we are not recursing on  $G_i$  (that is, we are not touching  $G_i$  again), the total cost of separating all such subgraphs over all levels of the recursion adds up to  $O(\text{sort}(N))$  in total. The separator vertices are the vertices of the  $O(X)$  cutter levels (each cutter level has at most  $Y = \frac{N}{\sqrt{R}}$  vertices), the separator vertices resulting from applying Lemma 3 to the subgraphs of bounded height and the separator vertices resulted from the recursive calls. Thus the total number of separator vertices is given by  $S(N) \leq X \frac{N}{\sqrt{R}} + \frac{N}{\sqrt{R}} + X \cdot S(\frac{N}{X})$ . If we choose  $X = (\frac{M}{B^2})^{1/4}$  and assume  $M > B^{2+\varepsilon}$ , for some  $\varepsilon > 0$ , it can be shown that  $X \frac{N}{\sqrt{R}} = O(\frac{N}{B})$  and  $\log_X \frac{N}{R} = O(\log_{M/B} \frac{N}{B})$ , so that  $S(N) = O(\text{sort}(N))$ .

The only thing remaining to discuss is how to represent a subgraph  $G_i$  between two cutter levels  $c_i$  and  $c_{i+1}$  in the format needed in order to apply Lemma 3 or perform the recursive call. Both these steps require that a BFS tree is given along with the subgraph. The part of  $T$  included in  $G_i$  is not connected and thus it is not a BFS tree for  $G_i$ . However, we can easily produce such a tree by introducing a “fake” root  $v_i$  and connecting it with “fake” edges to all vertices on level  $c_{i+1}$ . Note that if  $T$  is given level-by-level this can easily be done for all the subgraphs in  $O(\frac{N}{B})$  I/Os. The fake vertices and edges are marked so that they can be removed at the end of the algorithm. Details will appear in the full paper.

That our algorithm uses  $O(\text{sort}(N))$  I/Os can be seen as follows. The preprocessing step of computing the BFS level for each vertex in  $T$  and sorting the edges of  $G$  by level can easily be performed in  $O(\text{sort}(N))$  I/Os using standard techniques (such as list ranking and Euler tours) [12]. If we do not count the I/Os used to separate the subgraphs with bounded height, one recursion step can be performed in  $O(\frac{N}{B})$  I/Os, and the recurrence for the number of I/Os used becomes  $T(N) \leq \frac{N}{B} + X \cdot T(\frac{N}{X})$ . Thus  $T(N) = O(\text{sort}(N))$ . As the total number

As mentioned, the idea in our algorithm is to apply Lemma 3 to the subgraphs of bounded height  $O(\frac{N}{Y})$  and recursively separate the subgraphs of size  $O(\frac{N}{X})$ . By choosing  $Y = \frac{N}{\sqrt{R}}$  each bounded height subgraph  $G_i$  of size  $N_i$  has height  $\sqrt{R}$ , and it can thus be separated into  $\Theta(\frac{N_i}{R})$   $O(R)$

of I/Os used to separate the subgraphs of bounded height is  $O(\text{sort}(N))$ , we have shown that our algorithm uses  $O(\text{sort}(N))$  I/Os in total. This concludes the proof of Theorem 2.

So far we have only discussed the case  $R = \Omega(M)$ . If  $R$  is  $o(M)$  then we can use Theorem 2 to separate  $G$  in subgraphs of size  $O(M)$ , then load each subgraph into main memory one at a time and apply Lipton and Tarjan planar separator algorithm [23] until all subgraphs have size  $O(R)$ . This results in  $O(\frac{N}{\sqrt{R}})$  separator vertices. In some applications of the graph separation it is necessary to bound not only the total number of separators  $S$ , but also the number of separator vertices adjacent to any subgraph. This can be done as follows: For each subgraph which has  $\Omega(\frac{S}{N/R})$  adjacent separator vertices mark the inner vertices as inactive and apply Theorem 2 until the resulting subgraphs have  $O(\frac{S}{N/R})$  (active) vertices. Fredrickson [17] proves that this maintains the same bounds for the number of subgraphs and separators given that the graph has bounded degree. Details will appear in the full paper.

**Corollary 1.** *Let  $G = (V, E)$  be a planar graph and  $T$  a breadth-first search tree for  $G$ . Furthermore assume  $\exists \varepsilon > 0$  such that  $M > B^{2+\varepsilon}$ . Then  $G$  can be separated in  $\Theta(\frac{N}{R})$  subgraphs of  $O(R)$  vertices each and a set  $S$  of  $O(\text{sort}(N) + \frac{N}{\sqrt{R}})$  separator vertices using  $O(\text{sort}(N))$  I/Os.*

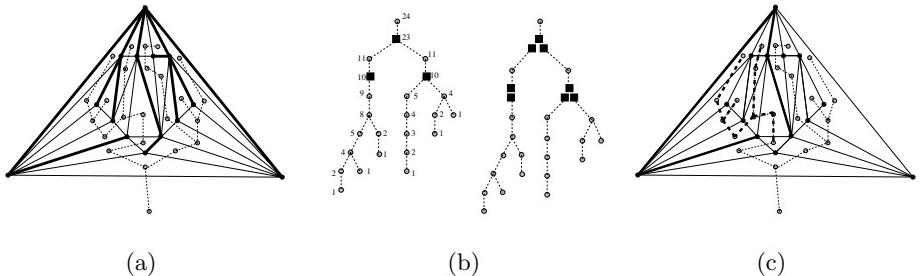
*If  $G$  has bounded degree then the separation can be constructed such that each subgraph  $G_i$  is adjacent to  $O(\frac{SR}{N})$  separator vertices.*

### 3.2 Separating Planar Graphs of Bounded Height Spanning Tree

In this section describe how we can separate in  $O(\text{sort}(N))$  I/Os a planar graph  $G = (V, E)$  with a spanning tree  $T$  of height  $H$  into  $\Theta(\frac{N}{R})$  subgraphs of size  $O(R)$  each and  $O(\frac{N}{R}H)$  separator vertices.

Assume for simplicity that  $G$  is triangulated. (If this is not the case, we can triangulate it using  $O(\text{sort}(N))$  I/Os [19] and mark the added edges so that they can be removed at the end of the separation. Note that  $T$  remains a spanning tree after the triangulation). Let  $G^*$  be the dual of  $G$  and let  $T^\dagger = (E - T)^*$  be the spanning tree in  $G^*$ . The spanning tree  $T^\dagger$  can be computed from  $G$  and  $T$  in  $O(\text{sort}(N))$  I/Os using a face finding algorithm as in [19] and a few sorting steps. Each edge in  $T^\dagger$  is the dual of an edge  $e = (u, v)$  in  $(E - T)$  and there exists a unique path from  $u$  to  $v$  in  $T$ ; this path and  $e$  forms a cycle in  $G$ , and since  $T$  has bounded height  $H$ , the cycle contains at most  $2H - 1$  vertices. Thus each edge in  $T^\dagger$  determines a cycle of size  $O(H)$  in  $G$  which separates  $G$  into the vertices inside the cycle and vertices outside the cycle. Refer to Fig. 2 (a). It can be shown that if  $e$  is the centroid edge of  $T^\dagger$ , then the number of vertices inside and outside the cycle is roughly the same [18].

The main idea in our algorithm is to find  $O(\frac{N}{R})$  cycles which partition  $G$  into subgraphs of roughly equal size  $O(R)$ . In order to do so, we first discuss how to find  $O(\frac{N}{R})$  edges in  $T^\dagger$  such that their removal divides  $T^\dagger$  into subtrees of roughly equal size  $O(R)$ . Then we show that the duals of these edges define  $O(\frac{N}{R})$  cycles in  $G$  with the desired properties.



**Fig. 2.** (a) A triangulated graph  $G$  (solid lines),  $T$  (solid thick lines) and  $T^\dagger$  (dotted lines). (b) The decomposition of  $T^\dagger$  into its 10-bridges; square vertices are the attachments. (c) Subtree of  $T^\dagger$  and the induced cycle in  $G$ .

The decomposition of a tree into independent subtrees of approximately equal size was studied by Gazit *et al.* [27] in the context of parallel  $R$ -contractions. We review briefly their notations and results. Let  $D = (V, E)$  be a tree with  $N$  vertices. The weight  $W(v)$  of a vertex  $v$  in  $D$  is the number of vertices in the subtree rooted at  $v$ . A vertex  $v$  is called *R-critical* if  $v$  is not a leaf and  $\lceil \frac{W(v)}{R} \rceil > \lceil \frac{W(v')}{R} \rceil$  for all children  $v'$  of  $v$ . Let  $C \subset V$ . Two edges  $e$  and  $e'$  of  $G$  are *C-equivalent* if there exists a path from  $e$  to  $e'$  that avoids the vertices  $C$ . The graphs induced by the equivalence classes of the  $C$ -equivalent edges are called the *bridges* of  $C$ . The *attachments* of a bridge  $I$  are the vertices of  $I$  that are also in  $C$ . The *R-bridges* of a tree  $D$  are the bridges of  $C$ , where  $C$  is the set of  $R$ -critical vertices of  $D$ . An example of the decomposition of a tree into its  $R$ -bridges is shown in Fig. 2 (b). Gazit *et al.* [27] prove the following: (1) The number of  $R$ -critical vertices in a tree of size  $N$  is at most  $\frac{2N}{R} - 1$ . (2) The number of  $R$ -bridges in a tree with bounded degree  $d$  is at most  $d(\frac{2N}{R} - 1)$ . (3) The number of vertices of an  $R$ -bridge is at most  $R + 1$ . (4) If  $I$  is an  $R$ -bridge, then  $I$  can have at most two attachments.

As the basic step in the computation of the  $R$ -bridges of  $D$  is the computation of the weight of each vertex, it is easy to show how standard I/O-efficient algorithms can be used to compute the  $R$ -bridges in  $O(\text{sort}(N))$  I/Os. If  $G$  is a triangulated graph,  $T^\dagger$  is a binary tree, and thus it has at most  $\frac{4N}{R}$   $R$ -bridges. Each  $R$ -bridge defines two cycles in  $G$  determined by the two edges incident to the two attachments. One of these cycles will be inside the other and there are at most  $R + 1$  faces inside the outer cycle but outside the inner cycle (the faces corresponding to the vertices in the  $R$ -bridge). Thus the  $R$ -bridges of  $T^\dagger$  determine a separation of  $G$  into  $\frac{4N}{R}$  subgraphs of at most  $R$  vertices adjacent to  $O(\frac{N}{R}H)$  separator vertices in total. Given the  $R$ -bridges, the decomposition of  $G$  can be easily computed in  $O(\text{sort}(N))$  I/Os and Lemma 3 follows.

## 4 Single Source Shortest Paths on Planar Graphs

In this section we show how to use our graph separation result to obtain an efficient SSSP algorithm for planar graphs with bounded degree.<sup>3</sup>

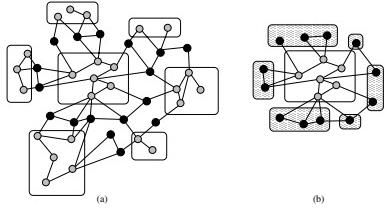
Consider separating a planar graph

$G$  into  $\Theta(\frac{N}{R})$  subgraphs  $G_i = (V_i, E_i)$  of  $O(R)$  vertices each and a set  $S$  of separator vertices, such that each subgraph is adjacent to  $O(\frac{SR}{N})$  separator vertices. We call the separator vertices adjacent to  $G_i$  the *boundary vertices* of  $G_i$ . Our algorithm relies on the following observation: Consider a shortest path  $\delta(s, t)$  between two vertices  $s$  and  $t$  in  $G$  and let  $\{s_0, s_1, \dots\}$  denote its intersection with  $S$ . The portion of  $\delta(s, t)$  between  $s_i$  and  $s_{i+1}$  is completely within some subgraph  $G_j$  and it must be the shortest path between  $s_i$  and  $s_{i+1}$  within  $G_j$ .

The main idea in our algorithm is to construct a new graph  $G^R$  by replacing each subgraph  $G_i$  with a complete graph on its boundary vertices. If the source vertex  $s$  is not a separator vertex, we also include  $s$  in  $G^R$  and connect it to the boundary vertices of the subgraph containing it. The graph  $G^R$  has  $S$  vertices and  $O(\frac{N}{R} \cdot (\frac{SR}{N})^2) = O(\frac{S^2 R}{N})$  edges. The weight of an edge in  $G^R$  is the length of the shortest path in  $G_i$  between the corresponding two boundary vertices. If  $R = O(M)$  these weights can be computed as follows: We load each subgraph  $G_i$  into main memory together with its boundary vertices and use an internal memory all-pair-shortest-paths algorithm to compute the weights of the new edges between the boundary vertices of  $G_i$ , and write these edges to the disk. Since each separator vertex is a boundary vertex for at most  $O(1)$  subgraphs (because of the bounded degree), we use at most  $(\frac{N}{B} + S)$  I/Os to load all the subgraphs and their boundary vertices. As we use  $O(\text{scan}(\frac{S^2 R}{N}))$  I/Os to write the new edges, it follows that  $G^R$  can be computed in  $O(S + \text{scan}(\frac{S^2 R}{N}))$  I/Os in total. Using  $S = O(\text{sort}(N) + \frac{N}{\sqrt{R}})$  (Corollary II) and choosing  $R = \frac{N^2}{\text{sort}^2(N)} = \frac{B^2}{\log^2_{M/B} N/B} < M$ , this is  $O(\text{sort}(N))$  I/Os.

Now assume we know how to compute the shortest paths from  $s$  to all separator vertices in  $O(\text{sort}(N))$  I/Os. Using the observation mentioned above, we know that these paths are identical to the shortest paths in the original graph  $G$ . We can then compute the shortest paths from  $s$  to all the remaining vertices in  $G$  by loading each subgraph  $G_i$  and its boundary vertices in main memory, and using an internal memory algorithm to compute the shortest path from  $s$  to each vertex  $t$  in  $V_i$  using the formula  $\delta(s, t) = \min_v \{\delta(s, v) + \delta_{G_i}(v, t)\}$ , where  $v$  ranges over all boundary vertices of  $G_i$ . This takes  $O(S + \text{scan}(N))$  I/Os, so the total number of I/Os used is  $O(\text{sort}(N))$ .

<sup>3</sup> Note that any graph can be transformed into a graph with each vertex having degree at most 3 using a simple transformation [17].



**Fig. 3.** (a) Separation of a graph into subgraphs (boxed) and separators (black); (b) a subgraph in the partition, its boundary vertices and boundary sets.

All that remains is to show how to solve the SSSP problem on the graph  $G^R$  with  $S = O(\text{sort}(N))$  vertices and  $O(\frac{S^2 R}{N}) = O(N)$  edges in  $O(\text{sort}(N))$  I/Os. To do so we use a slightly modified version of Dijkstra's algorithm which avoids the use of a *decrease\_key* priority queue operation. We want to avoid such an operation since the I/O bound of the best known external data structure with this operation is  $O(\frac{\log_2 N}{B})$  [22], while priority queues with  $O(\frac{\log_{M/B} N/B}{B})$  I/O bound are known if this operation is not supported [59]. During the algorithm we maintain a list  $L$  of pairs of vertices of  $G^R$  and their distances. Initially all distances are  $\infty$ . We maintain the invariant that the distance of a vertex in  $L$  is identical to the distances stored in the priority queue controlling the algorithm. The algorithm repeatedly performs a *delete\_min* operation on the priority queue to obtain the next vertex  $v$  to process; then the  $O(\frac{SR}{N}) = O(\frac{B}{\log_{M/B} N/B})$  edges incident to  $v$  are loaded using  $O(1)$  I/Os and the  $O(\frac{SR}{N}) = O(\frac{B}{\log_{M/B} N/B})$  boundary vertices adjacent to  $v$  are determined. These vertices (and their current distances) are loaded from  $L$  using  $O(\frac{B}{\log_{M/B} N/B})$  I/Os, and, without further I/Os we then compute which vertices need to have their distances updated. Finally, the new distances are written back to  $L$  and the corresponding updates are performed on the priority queue. Note that as we know the current distance of a vertex which needs to have its distance updated, we can perform the update in  $O(\frac{\log_{M/B} N/B}{B})$  I/Os using a *delete* and an *insert* operation.

Our algorithm performs  $O(N)$  operations on the priority queue using  $O(\text{sort}(N))$  I/Os in total. It also uses  $O(S) = O(\text{sort}(N))$  I/Os in total to load the neighbors of each vertex. Thus the I/O use is dominated by the  $O(\frac{B}{\log_{M/B} N/B})$  I/Os used for each vertex to load its adjacent vertices from  $L$ . Since there are  $O(\text{sort}(N))$  vertices, this sums up to  $O(\frac{B}{\log_{M/B} N/B}) \cdot O(\text{sort}(N)) = O(N)$  I/Os in total.

In order to improve the I/O bound to  $O(\text{sort}(N))$  we modify the algorithm, taking into account that there is some implicit adjacency between the boundary vertices. Let a *boundary set* be a maximal subset of boundary vertices such that all boundary vertices in the subset are adjacent to exactly the same subgraphs. An example is shown in Fig. 3 (b). Fredrickson [17] showed that the number of boundary sets is equal to the number of subgraphs  $O(\frac{N}{R})$ . We therefore modify our algorithm such that the vertices in the same boundary sets are stored consecutively in  $L$ . Otherwise the algorithm remains unmodified. When a vertex  $v$  is processed, the relevant boundary sets are determined and loaded from  $L$  as before. However, now we can think of the accesses as involving full boundary sets, as opposed to boundary vertices. Each boundary set is accessed  $O(\frac{B}{\log_{M/B} \frac{N}{B}})$  times (once by each of its adjacent boundary vertices), and as there are  $O(\frac{N}{R})$  boundary sets we use  $O(\frac{B}{\log_{M/B} \frac{N}{B}} \cdot \frac{N}{R}) = O(\text{sort}(N))$  I/Os in total.

**Theorem 3.** *Let  $G$  be a bounded degree planar graph and  $T$  a BFS tree for  $G$ . Furthermore assume  $\exists \varepsilon > 0$  such that  $M > B^{2+\varepsilon}$ . The SSSP problem on  $G$  can be solved in  $O(\text{sort}(N))$  I/Os.*

## References

1. J. Abello, A. L. Buchsbaum, and J. R. Westbrook. A functional approach to external graph algorithms. In *Proc. Annual European Symposium on Algorithms, LNCS 1461*, pages 332–343, 1998.
2. P. K. Agarwal, L. Arge, T. M. Murali, K. Varadarajan, and J. S. Vitter. I/O-efficient algorithms for contour line extraction and planar graph blocking. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 117–126, 1998.
3. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
4. ARC/INFO. *Understanding GIS—the ARC/INFO method*. ARC/INFO, 1993. Rev. 6 for workstations.
5. L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995. A complete version appears as BRICS technical report RS-96-28, University of Aarhus.
6. L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1004*, pages 82–91, 1995. A complete version appears as BRICS technical report RS-96-29, University of Aarhus.
7. L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. In *Proc. Workshop on Algorithm Engineering and Experiments*, 2000.
8. O. Boruvka. O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti*, 3:37–58, 1926.
9. G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1432*, pages 107–118, 1998.
10. A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 859–860, 2000.
11. A. L. Buchsbaum and J. R. Westbrook. Maintaining hierarchical graph views. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 566–575, 2000.
12. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.
13. F. Chin, J. Lam, and I. Chen. Efficient parallel algorithms for some graph problems. *Communications of ACM*, 1982.
14. R. Cole and U. Vishkin. Approximate parallel scheduling. II. Applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92(1):1–47, May 1991.
15. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Mass., 1990.
16. E. Feuerstein and A. Marchetti-Spaccamela. Memory paging for connectivity and path problems in graphs. *LNCS*, 762:416–425, 1993.
17. G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal of Computing*, 16:1004–1022, 1987.
18. M. Goodrich. Planar separators and parallel polygon triangulation. *Journal of Computer and System Sciences*, 51(3):374–389, 1995.

19. D. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. In *Proc. 5th Annual Int. Conf. Computing and Combinatorics*, number 1627 in LNCS. Springer-Verlag, July 1999.
20. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.
21. D. Kozen. *The Design and Analysis of Algorithms*. Springer, Berlin, 1992.
22. V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, pages 169–177, 1996.
23. R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal of Applied Math.*, 36:177–189, 1979.
24. A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. Manuscript, 1999.
25. K. Munagala and A. Ranade. I/O-complexity of graph algorithm. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 687–694, 1999.
26. M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.
27. J. H. Reif, editor. *Synthesis of Parallel Algorithms*, chapter 3, pages 115–194. Morgan Kaufmann, 1993.
28. R. E. Tarjan. *Data structures and network algorithms*. SIAM, Philadelphia, 1983.
29. J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intelligence*, 3:331–360, 1991.
30. J. S. Vitter. External memory algorithms (invited tutorial). In *Proc. of the 1998 ACM Symposium on Principles of Database Systems*, pages 119–128, 1998.

# I/O-Space Trade-Offs

## (Extended Abstract)

Lars Arge<sup>\*,1</sup> and Jakob Pagter<sup>\*\*,2</sup>

<sup>1</sup> Department of Computer Science, Duke University, Durham, NC 27708, USA

<sup>2</sup> BRICS<sup>\*\*\*</sup>, University of Aarhus, DK-8000 Aarhus C, Denmark

**Abstract.** We define external memory (or I/O) models which capture space complexity and develop a general technique for deriving I/O-space trade-offs in these models from internal memory model time-space trade-offs. Using this technique we show strong I/O-space product lower bounds for SORTING and ELEMENT DISTINCTNESS. We also develop new space efficient external memory SORTING algorithms.

## 1 Introduction

In internal memory models the time and space complexity, as well as trade-offs between the two, are well studied for fundamental problems such as SORTING and ELEMENT DISTINCTNESS. For example, Pagter and Rauhe [20] recently proved an  $O(N^2)$  upper bound on the time-space product for SORTING  $N$  elements, matching a lower bound of Beame [9]. Their algorithm can be used to improve space usage of time-optimal internal memory SORTING by a factor of  $\Theta(\log^2 N)$  compared to classical algorithms like MERGESORT and HEAPSORT. Such an improvement would be of considerable practical interest when dealing with massive data sets residing on external storage devices such as disks. For example, if dealing with 50GB of data even a factor of  $\log N$  would amount to a space reduction of a factor of more than 30. Unfortunately, very little is known about space complexity in external memory models where the main complexity measure is the number of I/Os needed to solve a problem. For example, even though several I/O-optimal external SORTING algorithms have been developed, no algorithm using sub-linear disk space—not counting the (read only) space holding the input—is known. One reason for this is that no external memory model capturing sub-linear space complexity has been defined.

In this paper we define external memory models which capture space complexity and use them to study I/O-space trade-offs for fundamental problems such as SORTING and ELEMENT DISTINCTNESS.

\* Supported in part by National Science Foundation ESS grant EIA-9870734, RI grant EIA-997287, and CAREER grant EIA-9984099. E-mail: [large@cs.duke.edu](mailto:large@cs.duke.edu).

\*\* Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT). Parts of this work was done while the author was visiting Duke University and while visiting University of Toronto. E-mail: [pagter@brics.dk](mailto:pagter@brics.dk).

\*\*\* Basic Research in Computer Science. Centre of the Danish National Research Foundation.

## 1.1 Related Work

The study of internal memory time-space trade-offs attempts to give formulae that relate time  $T$  and space  $S$  for a given problem. A typical result is of the form  $T \cdot S = O(f(N))$  for some problem and function  $f$ . Time-space trade-offs for **SORTING** are well studied [9,10,12,13,19,20], and for time above (roughly)  $N \log_2 N$  the exact complexity has been established to be  $T \cdot S = \Theta(N^2)$  [9,20]. This means that if  $S = \Omega(N / \log_2 N)$  then we can sort in time  $O(N \log_2 N)$ , and this is the best possible. Similarly, if  $T = O(N^{3/2})$  then  $S = \Theta(N^{1/2})$  is possible and required. Time-space trade-offs for **ELEMENT DISTINCTNESS** have also been studied extensively [3,11,15].

The standard model for studying external memory algorithms (or I/O-algorithms) is the I/O-model of Aggarwal and Vitter [2]. In this model the internal memory of size  $M$  is divided into  $m = M/B$  blocks of size  $B$  each. The external memory is also divided into blocks of size  $B$ , and initially the  $N$  input data elements reside in the first  $n = N/B$  blocks of external memory. An I/O is the movement of one block of elements to or from external memory, and the goal is to design algorithms that use as few I/Os as possible under the constraint that computation can only be performed on elements in internal memory.

During the last decade, a large number of I/O-efficient algorithms have been developed in the I/O-model—see e.g. recent surveys [5,23]. For example, it is well known that (under some restrictions) **SORTING** requires  $\Omega(n \log_m n)$  I/Os and several  $O(n \log_m n)$  algorithms using  $O(n)$  extra space (disk blocks) have been developed. However, no results are known about I/O-space trade-offs for **SORTING**. Similarly for **ELEMENT DISTINCTNESS**, Arge et al. [6] showed that the problem is as hard as **SORTING** in a comparison based model, and Arge and Miltersen [7] gave an  $O(n)$  I/O (and space) randomized algorithm for the problem, but nothing is known with respect to I/O-space trade-offs.

It should be mentioned that one reason no I/O-space trade-offs are known in the I/O-model is that it allows for the input to be overwritten. Thus  $\Omega(n)$  space is always available for algorithms in the model and one cannot formally express sub-linear space bounds. External memory models that capture space complexity have been introduced in the area of straight-line computation models (i.e., models where branching or “if-then-else” statements are not allowed). An example is the so-called red-blue pebble games—see e.g. [14,21]. However, disallowing branching is too severe a restriction for our purposes.

## 1.2 Our Results

In Section 2 we introduce computational models which allow us to study I/O-space trade-offs. We first introduce an extension of the Aggarwal and Vitter model (or rather an I/O-version of the RAM-model equivalent to their model) in which sub-linear space complexity can be expressed. This is the model we will use when developing space-efficient I/O-algorithms. We next extend the branching program model [10,12,21]—the model most commonly used for showing internal memory time-space trade-offs—in order to capture I/O and space complexity

simultaneously. We also prove that lower bounds in this model are valid in the extended I/O-model.

In Section 3 we develop a technique for obtaining I/O-space trade-offs for external memory computation from time-space trade-offs for internal memory computation. More precisely, let  $T$  and  $S$  denote the time and space usage of an internal memory algorithm solving a problem  $P$  over  $U = \{1, \dots, R\}$ . Similarly, let  $T^{IO}$  denote the number of I/Os and  $S^{IO}$  the space usage of an external memory algorithm solving  $P$ . We prove that if  $T = \Omega(f(N, S))$ , then  $T^{IO} = \Omega(\frac{f(N, S^{IO})}{B})$ .

Using the general result we prove I/O-space trade-offs for SORTING and ELEMENT DISTINCTNESS. Combining the result with a result of Beame [9], we for example show that for SORTING the I/O-space product is  $\Omega(N^2/B) = \Omega(N \cdot n)$ . Using the internal memory algorithm of Pagter and Rauhe [20] in external memory shows that this bound is tight among algorithms using more than (roughly)  $N \log_2 N$  I/Os. This is an interesting result, as it suggests that when disk space is restricted, traditional internal memory approaches can lead to optimal external memory algorithms. The results for ELEMENT DISTINCTNESS are obtained by applying our results to the internal memory lower bounds of Ajtai [3] and Yao [24].

Finally in Section 4 we discuss an external memory generalization of the algorithm of Pagter and Rauhe which for certain choices of  $M$  and  $B$  obtains the optimal I/O-space trade-off for SORTING down to the optimal number of I/Os  $O(n \log_m n)$ . In general however, we can only prove an  $O((N^2/(B+m)) \log_2 m)$  upper bound on the I/O-space product. We conjecture that our lower bound for SORTING is tight for all values of  $M$  and  $B$ , that is, an algorithm achieving  $O(N^2/B)$  exists.

## 2 Models of Computation

In this section we introduce computational models which allows us to study I/O-space trade-offs. In Section 2.1 we first consider upper bound models and in Section 2.2 we then consider lower bound models. We discuss the relationship between the models in Section 2.3.

One main difference between our models and the standard I/O-model is that we assume input to be read-only. A natural question is whether this is reasonable and we claim that it is. Consider for example the task of SORTING a huge database by a secondary key: In such an example it might be important not to overwrite the original database sorted by primary key. A typical example is the customer database of a bank, which will normally be sorted by account numbers. Occasionally the bank might want a phone book over its customers, requiring the database to be sorted on customer names, but rarely will it be interested in erasing the original database used for all standard business transactions. Other examples occur when the input is stored on a medium which is physically read-only, for example on a CD-ROM.

## 2.1 Upper Bound Models

In this section we define two external memory extensions of the unit cost RAM model. The unit cost RAM model is a popular internal memory model for showing upper bounds. In this model input consists of  $N$  words  $x_1, \dots, x_N$  from some universe  $U = \{0, 1\}^w$ , and it is a normal convention that each word in memory can hold exactly  $w$  bits, corresponding to one input element. We use this convention and define a parameter  $R = 2^w$  (i.e. each input element can represent one of  $R$  values). We will always assume that  $R \geq N$ .

**Definition 1 (External RAM)** *The external RAM consist of two parts: A memory layout, and an instruction set:*

**Memory layout:** *Input is located on a read-only input medium consisting of  $N$  words of  $\log_2 R$  bits, grouped into  $n = N/B$  consecutive blocks of  $B$  words. Output is to be written to a separate write-only output medium. Furthermore, we have an external memory consisting of blocks of  $B$  words of  $\log_2 R$  bits each, and an internal memory consisting of  $M/B$  blocks of  $B$  words of  $\log_2 R$  bits each.*

**Instructions:** *The algorithm can execute the following instructions:*

- 1 *Read a block from the input into internal memory*
- 2 *Write a block from internal memory to the output*
- 3 *Swap a block from internal memory with one from external memory (both may be “empty”)*
- 4 *Perform some unit-cost operation on two words in internal memory, writing the result to one word in internal memory.*

□

The number of I/Os  $T_R^{IO}$  performed by an external RAM is the number of times the algorithm executes instructions 1, 2 or 3, and we define space  $S_R^{IO}$  to be the number of bits occupied by the external memory. Note that we ignore the  $M \log_2 R$  bits available in internal memory since we will always assume that  $S_R^{IO} \geq M \log_2 R$ . Also note that we define space in terms of bits and not words or blocks. One can of course easily translate our space measure into measures based on words ( $S_R^{IO} / \log_2 R$ ) or blocks ( $S_R^{IO} / (B \log_2 R)$ ). In the introduction the space bounds were expressed in terms of blocks.

It should be clear that the external RAM model is essentially a straightforward extension of the I/O-model of Aggarwal and Vitter—basically the input has just been made read-only.

**Definition 2 (Comparison external RAM)** *A comparison external RAM is an external RAM, with instruction 4 replaced by*

- 4a *Compare input elements, or copies thereof, in two words in internal memory using a binary comparison*
- 4b *Perform some unit-cost operation on two words in internal memory which are not occupied by input elements, or copies thereof, writing the result to one word in internal memory*

□

The number of I/Os  $T_C^{IO}$  performed by a comparison external RAM is the number of times the algorithm executes instructions 1, 2 or 3, and the space use  $S_C^{IO}$  is the number of bits occupied by the external memory.

Note that in a comparison external RAM one can *only* access words containing input elements, or copies thereof, through comparisons. All other words in internal memory can be manipulated freely. We think of each word containing (a copy of) an input element as being marked, and such marked words can only be accessed via comparisons with another marked word. This somewhat strange way of enforcing a comparison model is a result of the fact that in an external memory model it is vital that we are allowed to make and move copies of elements (since block movement is the main complexity measure). Another standard way of enforcing a comparison model is to ensure that elements are indivisible and that new elements cannot be produced, that is, that each word in memory is either empty or contains an input element (the so-called *indivisibility assumption* [2,7]). However, there is evidence that the indivisibility assumption drastically increases the complexity of certain problems [17], and furthermore, we would like to be able to manipulate objects such as pointers in our algorithms.

It should be clear that the external RAM models can easily simulate algorithms constructed for models where one is allowed to overwrite the input: Simply make a copy of the input in the external memory and run the algorithm on this copy. In particular, the I/O-optimal  $O(n \log_m n)$  SORTING algorithms of Aggarwal and Vitter [2] may be implemented on the comparison external RAM. Note also that the external RAM is at least as strong as the comparison external RAM.

## 2.2 Lower Bound Models

The models we will use for showing lower bounds (on I/O-space trade-offs) are extensions of the branching program model. The branching program model is a well established internal memory model for showing lower bounds on time-space complexity, see e.g. [3,9,10,11,12,15,17,24]. Branching programs come in two main variants: Comparison based branching programs, which were initially studied in [22] according to which they were introduced by Pippenger, and  $R$ -way branching programs introduced by Borodin and Cook [10]. Detailed discussions of branching programs can be found in [12,21].

A *comparison branching program* is a directed acyclic graph (DAG) with one root. Each non-leaf node is labeled  $(i : j)$  and has two outgoing arcs labeled  $x_i < x_j$  and  $x_i \geq x_j$ . A computation starts at the root and proceeds in the natural way until a leaf is reached. If one is studying decision problems, each leaf will be labeled 0 or 1 depending on whether the corresponding computation rejects or accepts the input. For functions such as SORTING, each arc may be further labeled with elements from the output domain. The output of the computation is the ordered concatenation of the outputs encountered along the computation path.

Time  $T_C$  in the comparison branching program model is defined as the height of the branching program, corresponding to the number of comparisons per-

formed in the worst case, and space  $S_C$  is defined as  $\log_2 |V|$ , where  $V$  is the set of vertices of the branching program. This is an adequate space measure since it gives a lower bound on the number of bits required to distinguish between the states of the program. We will discuss this in greater detail when comparing RAM models and branching programs in Section 2.3.

An  $R$ -way branching program is a DAG with one root, where each non-leaf (branching) node is labeled with an index  $i$  ( $1 \leq i \leq N$ ) and has an outgoing arcs for each element of  $U = \{0, 1\}^w$  (recall that  $R = 2^w$ ). A computation proceeds as before except that in a node labeled  $i$  the arc labeled  $l$  is followed if  $x_i = l$ ; i.e., a branch is made according to the value of  $x_i$ .

As previously,  $T_R$  is defined as the height of the DAG, corresponding to the number of times we read one of the elements from the input in the worst case. Space  $S_R$  is defined as for comparison branching programs. As we can simulate a comparison using two  $R$ -way branches,  $R$ -way branching programs are asymptotically stronger than comparison branching programs (for  $R = N^{O(1)}$ ). From a practical point of view, the  $R$ -way branching program model is an unrealistically strong model of computation; given enough space to remember the value of all the input elements— $O(R^N)$  nodes or space  $O(N \log_2 R)$ —one can decide any problem after reading each element *once*, i.e., in linear time. On the other hand, when restricting the space one can prove interesting and sometimes even tight time-space trade-offs in the model. In the following we define external versions of the branching program models.

**Definition 3 (Comparison external branching program)** *A comparison external branching program is a comparison branching program with two types of nodes—comparison nodes and I/O-nodes: An I/O-node replaces any  $B$  elements in the internal memory of size  $M$  with any  $B$  elements from the input. A comparison node can only compare two elements which are both in internal memory.*

□

The number of I/Os  $T_C^{IO}$  performed by a comparison external branching program is defined as the maximum number of I/O-nodes encountered along any root-leaf path. As previously, space  $S_C^{IO}$  is defined as  $\log_2 |V|$ , where  $V$  is the set of vertices of the branching program.

We emphasize the fact that one can read *any*  $B$  elements when making a block transfer (I/O) from input to internal memory. This seems like a strong and unrealistically powerful operation, but not only can we prove interesting—in some cases even tight—lower bounds in this model, we also (as we will see) need this strength in order to simulate comparison external RAM algorithms (Theorem 1). Note also that the term “internal memory” is somewhat misleading, as the  $M$  elements are not physically present in any kind of memory. At any point of the computation, the elements “in internal memory” are just the  $M$  input elements (out of the  $N$  possible elements) the external branching program is allowed to compare. Finally, note that comparison external branching programs *cannot* make copies of elements, since they can only access input elements through comparisons.

**Definition 4 (*R*-way external branching programs)** An *R*-way external branching program is an *R*-way branching program with two types of nodes—branching-nodes and I/O-nodes: An I/O-node replaces any  $B$  elements in the internal memory of size  $M$  with any  $B$  elements from the input. A branching-node performs a branch based on the value of one of the  $M$  input elements in internal memory.  $\square$

The number of I/Os  $T_R^{IO}$  performed by an *R*-way external memory branching program, as well as the space use  $S_R^{IO}$ , is defined as for the comparison external branching program. For both *R*-way external branching programs and comparison external branching programs we again use the natural assumption that  $S^{IO} \geq M \log_2 R$ .

We call a (comparison or *R*-way) external branching program where all nodes have in-degree at most 1 an *external tree*. Arge et al. [6] defined a model similar to a comparison external tree called the *I/O decision-tree*, and Munagala and Ranade [18] defined a model similar to an *R*-way external tree. In both models one is only allowed to read contiguous input elements. However, unlike external trees, both models contain a mechanism for rearranging (writing) the input elements.

A key property of (comparison or *R*-way) external branching programs is that removing the I/O-nodes results in a standard branching program. In this subsection we will discuss two other properties of external branching programs.

A branching program is called *leveled* if the nodes of the program can be partitioned into  $T$  classes  $V_1, V_2, \dots, V_T$  such that arcs emanating from nodes in  $V_i$  go to nodes in  $V_{i+1}$ . Pippenger proved that any standard branching programs of height  $T$  using space  $S$  can be transformed into a leveled branching program with height  $T + 1$  and using space less than  $2S$ —see e.g. Borodin et al. [12]. We say that an I/O-node  $w$  is the *immediate I/O-successor* of a node  $v$ , if  $w$  is the first I/O-node encountered on one of the paths from  $v$  to a leaf of the branching program. Note that  $v$  may have several immediate I/O-successors. We say that an external branching program is *I/O-leveled*, if the I/O-nodes can be partitioned into  $T^{IO}$  classes  $V_1, V_2, \dots, V_{T^{IO}}$  such that all I/O-nodes in  $V_i$  have their immediate I/O-successors in  $V_{i+1}$ . With a simple modification of Pippenger’s proof we can show the following.

**Lemma 1** Any (comparison or *R*-way) external branching program using  $T^{IO}$  I/Os and  $S^{IO}$  space can be transformed into an I/O-leveled external branching program solving the same problem using  $T^{IO} + 1$  I/Os and space less than  $2S^{IO}$ .

The *immediate I/O-ancestor* of a node  $v$  can be defined analogously to immediate I/O-successor. An external branching program is called *I/O-separated* if all internal nodes other than I/O-nodes have only one immediate I/O-ancestor.

**Lemma 2** Any I/O-leveled (comparison or *R*-way) external branching program using  $T^{IO}$  I/Os and  $S^{IO}$  space can be transformed into an I/O-separated and I/O-leveled external branching program solving the same problem using  $T^{IO}$  I/Os and less than  $2S^{IO}$  space.

**Proof:** The details of the proof appear in the full version of this paper [8].  $\square$

In the following, we will assume without loss of generality that all external branching programs are I/O-leveled and I/O-separated.

### 2.3 External RAM's vs. External Branching Programs

In this section we prove two theorems which allow us to prove lower bounds for external RAM's using the external branching program models.

**Theorem 1** *For any comparison external RAM algorithm solving a problem  $P$  in  $T_C^{IO}$  I/Os and space  $S_C^{IO} \geq M \log_2 R$  there exists a comparison external branching program solving  $P$  in  $T_C^{IO}$  I/Os and space at most  $2S_C^{IO}$ .*

**Proof:** The basic idea of the proof is to construct a branching program with a node for each state of the RAM algorithm and with arcs reflecting how the computation proceeds. However, there seems to be one major problem with this idea, namely that while the comparison external RAM can make copies of input elements and move them around in memory, a comparison external branching program can only compare input elements. However, as we will show below, this problem can be overcome using the comparison external branching programs (powerful) ability to move *any*  $B$  input elements into internal memory instead of just contiguous ones. We also use the fact that comparison external RAM's can only access (copies of) input elements using comparisons.

As the comparison external RAM algorithm uses  $S_C^{IO}$  bits in external memory and have  $M \log_2 R$  bits in internal memory, it has at most  $2^{S_C^{IO} + M \log_2 R} \leq 2^{2S_C^{IO}}$  distinct states. We split these states into three types based on the operation performed in a given state:

1. I/Os (instructions 1, 2, and 3).
2. Comparisons of (copies of) input elements (instruction 4a).
3. Other operations (on non-input elements) (instruction 4b).

After performing an operation in a given state the comparison external RAM algorithm proceed to a new state: In a type 1 state the algorithm performs an I/O and proceeds to *one* unique state. In a type 3 state the algorithm performs some operation and proceeds to *one* unique state. Note that a sequence of type 3 states are “straight-line” in the sense that the computation performed in the sequence only depends on the first state in the sequence. In a type 2 state the algorithm proceeds to *one out of two* unique states, depending on the outcome of the comparison.

We construct a comparison external branching program from the comparison external RAM as follows: For each state in the comparison external RAM we construct a node in the comparison external branching program, and we connect these nodes according to how computation proceeds from state to state. A comparison external branching program cannot contain nodes corresponding to states of type 3, but as these perform straight-line computation we can remove all such nodes/states by coalescing a node/state (or a sequence of them) into their first successor of type 1 or 2. Nodes corresponding to type 2 states can be

left unmodified, but nodes corresponding to type 1 states need to be modified since I/O-nodes in a comparison external branching program always read  $B$  input elements from input to internal memory, while much more complicated I/Os can be performed in type 1 states. Nodes corresponding to type 1 states which only *read from the input* we just leave unchanged. Nodes corresponding to type 1 states which *write to output* are removed and the arc between its predecessor and successor is labeled with the appropriate output information. Nodes corresponding to type 1 states which *swap a block* in internal memory with a block from external memory are more complicated since we can only read input elements in comparison external branching programs. Consider the block of  $B$  elements swapped into internal memory in such a node/state. Some of these elements are copies of the original input elements and some contain other information. The latter can be disregarded since the same information is represented in the state itself (recall that a state represent the content of the entire memory). We replace the node with a node which read the relevant less than  $B$  elements from the input. In general these input elements will not constitute a block of the input but we utilize that external branching programs can read any  $B$  input elements in one swap operation.

It should be clear that the comparison external branching program constructed in this way solves the problem  $P$  in  $T_C^{IO}$  I/Os. The branching program uses less than  $\log_2 2^{2S_C^{IO}} = 2S_C^{IO}$  space.  $\square$

We can prove a similar theorem for external RAM algorithms and  $R$ -way external branching programs. We omit the proof as it is very similar to the proof of Theorem 1.

**Theorem 2** *For any external RAM algorithm solving a problem  $P$  in  $T_R^{IO}$  I/Os and space  $S_R^{IO} \geq M \log_2 R$  there exists a  $R$ -way external branching program solving  $P$  in  $T_R^{IO}$  I/Os and space at most  $2S_R^{IO}$ .*

### 3 Lower Bounds

In this section we first present a general method for obtaining I/O-space trade-offs in external branching program models from time-space trade-offs in normal branching program models. Using this method we then obtain trade-offs for SORTING and ELEMENT DISTINCTNESS.

#### 3.1 General Lower Bound Method

In [6] Arge et al. describes a general technique for transforming an internal memory decision tree lower bound into an I/O-decision tree I/O lower bound (see also [718]). The main idea in their technique is a method for transforming an I/O-decision tree algorithm into an internal decision tree algorithm, such that the number of comparisons performed by the internal algorithm is bounded by a function of the number of I/Os performed by the external algorithm. In this section we apply the same idea to branching program models. We first consider the comparison model.

**Theorem 3** Suppose that for any comparison branching program solving a problem  $P$  in time  $T_C$  and space  $S_C$  we have  $T_C \geq f(N, S_C)$ . Then for any comparison external branching program solving  $P$

$$T_C^{IO} = \Omega\left(\frac{f(N, S_C^{IO})}{B \log_2 M}\right).$$

**Proof:** As mentioned, the proof is based on the ideas used by Arge et al. [6]; we will describe a general method for transforming a comparison external branching program solving  $P$  using  $T_C^{IO}$  I/Os and  $S_C^{IO}$  space into a comparison branching program solving  $P$ . The comparison branching program lower bound then results in a comparison external branching program lower bound.

Consider an I/O-leveled and I/O-separated comparison external branching program, and as previously let the sub-branching program rooted at an I/O-node  $v$  consist of all nodes reachable from  $v$  without passing through another I/O-node. Recall that as the comparison external branching program is I/O-leveled, each comparison-node is contained in precisely one sub-branching program. Our aim is to replace each sub-branching program with a new sub-branching program that computes the total ordering of all the elements in internal memory; since we only have comparison based access to the input, computing the total order means that any question regarding the input answered by the old sub-branching program can be answered by the new sub-branching program. We first transform the comparison external branching program into another comparison external branching program where we know the total order of the input elements in internal memory in each I/O-node. (Note that there are  $M!$  different orderings of the  $M$  elements in internal memory). In order to do so we first make  $M!$  copies of the original comparison external branching program—in our construction each of these copies will be used to represent a unique total order of the elements in internal memory. Each copy of the comparison external branching program is now transformed into another comparison external branching program I/O-level by I/O-level, top-down (while maintaining the invariant that the internal memory element order is known in I/O-nodes on already processed levels). We transform the sub-branching program rooted in I/O-node  $v$  as follows: We replace the sub-branching program with a sub-branching program which first computes the total order of the  $B$  input elements loaded into internal memory by  $v$  (i.e. it sorts them), and then finds the positions of the  $B$  elements among the sorted elements already in internal memory (i.e. it merges the  $B$  “new” elements with the  $M - B$  “old” elements). As shown in [6], the height of the new sub-branching program can be bounded by  $O(B \log_2 B + B \log_2 m) = O(B \log_2 M)$ , which in turn means that it contains  $O(2^{B \log_2 M})$  nodes. Each leaf  $l_n$  of the new sub-branching program corresponds to a total order of the elements in internal memory. Thus it also corresponds to a unique leaf  $l_o$  in the original sub-branching program, namely the leaf at which inputs with this particular total ordering would end up. To guarantee that the transformed comparison external branching program solves  $P$ , we want to connect  $l_n$  to the same I/O-node  $v_o$  on the next I/O-level as  $l_o$  is connected to. Note however, that several leaves (total orders) in the new

sub-branching program could correspond to  $l_o$ . In order not to lose information about the order of the elements in internal memory, we therefore connect  $l_n$  to  $v_o$  of the copy of the original program corresponding to the total order of the elements in internal memory. After processing all I/O-nodes on the same level as  $v$ , we go on and process the next I/O-level.

After the transformation we have a new comparison external branching program with the same overall structure as the original comparison external branching program, specifically the new branching program has at least the same knowledge about the input as the original branching program. Consequently the new branching program can answer any question that the original branching program answers and thus it solves problem  $P$ .

From the construction it should also be clear that the new comparison external branching program has the same I/O-height  $T_C^{IO}$  as the original comparison external branching program. That the space use of the new comparison external branching program is  $O(S_C^{IO})$  can be seen as follows: We have  $M!$  copies of the original program, each containing no more than  $|V| = 2^{S_C^{IO}}$  I/O-nodes. For each I/O-node we have a sub-branching program of size less than  $2^{B \log_2 M}$ . Thus in total the new comparison external branching program use space  $O(\log_2(M! \cdot 2^{S_C^{IO}} \cdot 2^{B \log_2 M})) = O(M \log_2 M + S_C^{IO} + B \log_2 M) = O(M \log_2 R + S_C^{IO}) = O(S_C^{IO})$ .

Next we simply remove the I/O-nodes from the transformed comparison external branching program and obtain a (standard) comparison branching program with height  $T_C = O(T_C^{IO} \cdot B \log_2 M)$  using space  $S_C = O(S_C^{IO})$ . The theorem follows since  $T_C \geq f(N, S_C)$ .  $\square$

We can prove a similar theorem for  $R$ -way external branching programs.

**Theorem 4** *Suppose that for any  $R$ -way branching program solving a problem  $P$  in time  $T_R$  and space  $S_R$  we have  $T_R \geq f(N, S_R)$ . Then for any  $R$ -way external branching programs solving  $P$*

$$T_R^{IO} = \Omega\left(\frac{f(N, S_R^{IO})}{B}\right).$$

**Proof (sketch):** We use the same proof technique (construction) as in the comparison model (Theorem 3). The  $B \log_2 M$  factor in the comparison model construction was a result of this being the height of the sub-branching program used to obtain full information about the elements in internal memory after an I/O. We will use the power of the  $R$ -way model to decrease this height and thus obtain an improved bound. In the  $R$ -way model, having full information about the elements in internal memory corresponds to knowing the value of all the elements. Thus the main idea in the  $R$ -way construction is to replace each sub-branching program with a sub-branching program which "remembers" the value of the  $B$  new elements in internal memory after an I/O. Since such a sub-branching program has height  $B$  (we just read each element once) the theorem follows. Details appear in the full version of this paper [8].  $\square$

Note the interesting and somewhat counterintuitive fact that by switching to a stronger model (from Theorem 3 to Theorem 4) we obtain a better lower bound.

### 3.2 Applications of the Lower Bound Method

In the  $R$ -way model, Beame [9] proved that  $T_R \cdot S_R = \Omega(N^2)$  for the problem of SORTING elements in the universe  $U = \{1, \dots, N^2\}$ . Using Theorem 4 we thus obtain the following.

**Corollary 1** *Any  $R$ -way external memory branching program SORTING  $N$  numbers from a universe  $U = \{1, \dots, N^2\}$  has  $T_R^{IO} \cdot S_R^{IO} = \Omega(\frac{N^2}{B})$ .*

Ajtai [3] proved that for ELEMENT DISTINCTNESS over  $U = \{1, \dots, N^2\}$  we have that if  $T_R = O(N)$  then  $S_R = \Omega(N)$ . Using Theorem 4 we obtain the following.

**Corollary 2** *Any  $R$ -way external memory branching program solving the ELEMENT DISTINCTNESS problem on  $N$  numbers from the universe  $U = \{1, \dots, N^2\}$  with  $T_R^{IO} = O(n)$  uses space  $S_R^{IO} = \Omega(N)$ .*

Corollary 1 implies (among other things) that we must use  $\Omega(N/(\log_m n))$  space in order to sort  $N$  elements in the optimal  $O(n \log_m n)$  I/Os. This should be compared to the  $N \log_2 R$  space use of the sorting algorithm of Aggarwal and Vitter [2]. Corollary 2 means that in order to decide ELEMENT DISTINCTNESS in a linear number of I/Os we must use at least linear extra space.

Yao [24] proved that any comparison branching program deciding ELEMENT DISTINCTNESS on  $N$  numbers must have  $T_C \cdot S_C = \Omega(N^{2-\epsilon(N)})$ , where  $\epsilon(N) = 5/\sqrt{\log_2 N}$ . Using Theorem 3 we thus obtain the following.

**Corollary 3** *Any comparison external branching program deciding ELEMENT DISTINCTNESS on  $N$  numbers has  $T_C^{IO} \cdot S_C^{IO} = \Omega(\frac{N^{2-\epsilon(N)}}{B \log_2 M})$ .*

## 4 Upper Bounds

In this section we briefly discuss our new upper bounds on the I/O-space product for SORTING. Details appear in the full version of this paper [8].

Modifying the internal memory SORTING algorithm of Pagter and Rauhe [20] in a straightforward way to make it work in external memory we can obtain the following.

**Theorem 5** *There exists positive constants  $c_1$  and  $c_2$  so that there exists a comparison external RAM algorithm SORTING  $N$  numbers in  $c_1 N \log_2 N \leq T_C^{IO} \leq c_2 N^2/(B \log_2 N)$  I/Os and space  $S_C^{IO}$  such that  $T_C^{IO} \cdot S_C^{IO} = O(\frac{N^2}{B})$ .*

It follows from Corollary 1 that Theorem 5 is optimal. Note that this means that traditional internal memory approaches can lead to optimal external memory algorithms when disk space is restricted.

In the more important case where  $n \log_m n \leq T_C^{IO} \leq N \log_2 N$  (roughly) we can sometimes also obtain optimal bounds. In order to do so we need to make

several modifications to the algorithm by Pagter and Rauhe. Their algorithm is based on a complicated binary tree data structure similar to a tournament tree (see e.g [16]). By designing a  $m$ -ary version of their data structure we first reduce  $T_C^{IO}$  to  $O(N^2/(kB) + N \log_2 m \cdot \log_m k)$  using space  $O(k \log_2 m)$ —i.e. we obtain a data structure where we may vary I/O and space usage by varying the parameter  $k$ . To improve this bound we utilize a variant of the buffering technique of Arge [4], as well as the fact that one can sort  $M$  elements in internal memory without performing any I/Os. Using these ideas we can further reduce  $T_C^{IO}$  to

$$O\left(\frac{\frac{N^2}{k(B+m)} + N \log_m k}{B}\right)$$

increasing the space use with a factor  $B$  to  $O(Bk \log_2 m)$ . By choosing  $k$  appropriately we then obtain the following.

**Theorem 6** *There exists positive constants  $c_1$  and  $c_2$  so that there exists an comparison external RAM algorithm SORTING  $N$  numbers in  $c_1 n \log_m n \leq T_C^{IO} \leq c_2 N \log_2 N$  I/Os and space  $S_C^{IO}$  such that  $T_C^{IO} \cdot S_C^{IO} = O(\frac{N^2 \log_2 m}{B+m})$ .*

Note that Theorem 6 means that if  $M \geq B^2$  (a realistic and standard assumption in the external memory literature) we can sort I/O-optimally (using  $O(n \log_m n)$  I/Os) using (sub-optimal) space  $O((N \log_2 m) / \log_m n)$ . In comparison, the SORTING algorithm of Aggarwal and Vitter [2] uses roughly a factor  $\log_2 n$  more space  $O(N \log_2 N)$ . If also  $\log_2 m \leq \log_m n$  we achieve optimal space. We conjecture that our lower bound for SORTING is tight for all values of  $M$  and  $B$ , that is, an algorithm achieving a I/O-space product of  $O(N^2/B)$  exists.

## Acknowledgments

We would like to thank an anonymous referee for valuable comments on presentation. The second author would like to thank Faith Fich.

## References

1. M. Adler. New coding techniques for improved bandwidth utilization. In *Proc. 37th Annual Symposium on Foundations of Computer Science*, pages 173–182, 1996.
2. A. Aggarwal and J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
3. M. Ajtai. Determinism versus Non-Determinism for Linear Time RAMs with Memory Restrictions. In *Proc. Thirty-First ACM Symposium on Theory of Computing*, 1999.
4. L. Arge. The Buffer Tree: A New Technique for Optimal I/O-Algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995. A complete version appears as BRICS technical report RS-96-28, University of Aarhus.
5. L. Arge. *Efficient External-Memory Data Structures and Applications*. PhD thesis, University of Aarhus, 1996.

6. L. Arge, M. Knudsen, and K. Larsen. A General Lower Bound on the I/O-Complexity of Comparison-based Algorithms. In *Proc. of the Workshop on Algorithms and Datastructures, LCNS 709*, pages 83–94, 1993.
7. L. Arge and P. B. Miltersen. On showing lower bounds for external-memory computational geometry problems. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. AMS Press, 1999.
8. L. Arge and J. Pagter. I/O-Space Trade-Offs. Technical Report BRICS-RS-00-7, BRICS, University of Aarhus, Denmark, April 2000. Available via [www.brics.dk](http://www.brics.dk).
9. P. Beame. A General Sequential Time-Space Tradeoff for Finding Unique Elements. *SIAM Journal on Computing*, 20:270–277, 1991.
10. A. Borodin and S. Cook. A Time-Space Tradeoff for Sorting on a General Sequential Model of Computation. *SIAM Journal on Computing*, 11(2):287–297, 1982.
11. A. Borodin, F. E. Fich, F. Meyer auf der Heide, E. Upfal, and A. Wigderson. A Time-Space Tradeoff for Element Distinctness. *SIAM Journal on Computing*, 16:97–99, 1987.
12. A. Borodin, M. J. Fischer, D. G. Kirkpatrick, N. A. Lynch, and M. Tompa. A Time-Space Tradeoff for Sorting on Non-Oblivious Machines. *Journal of Computer and System Sciences*, 22:351–364, 1981.
13. G. N. Frederickson. Upper Bounds for Time-Space Trade-offs in Sorting and Selection. *Journal of Computer and Systems Sciences*, 34:19–26, 1987.
14. J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. ACM Symp. on Theory of Computation*, pages 326–333, 1981.
15. M. Karchmer. Two Time-Space Tradeoffs for Element Distinctness. *Theoretical Computer Science*, 47:237–246, 1986.
16. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 2nd edition, 1998.
17. Y. Mansour, N. Nisan, and P. Tiwari. The Computational Complexity of Universal Hashing. *Theoretical Computer Science*, (107):121–133, 1993.
18. K. Munagala and A. Ranade. I/O-complexity of graph algorithm. In *Proc. Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694, 1999.
19. J. I. Munro and M. S. Paterson. Selection and Sorting with Limited Storage. *Theoretical Computer Science*, 12:315–323, 1980.
20. J. Pagter and T. Rauhe. Optimal Time-Space Trade-Offs for Sorting. In *proc. 39th Annual Symposium on Foundations of Computer Science*, pages 264–268, 1998.
21. J. E. Savage. *Models of Computation*. Addison-Wesley, 1998.
22. M. Tompa. Time-Space Tradeoffs for Straight-Line and Branching Programs. Technical Report 122/78, Department of Computer Science, University of Toronto, July 1978. Ph.D. Thesis.
23. J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*. AMS Press, 1999.
24. A. C.-C. Yao. Near-optimal Time-Space Tradeoff for Element Distinctness. *SIAM Journal on Computing*, 23:966–975, 1994.

# Optimal Flow Aggregation

Subhash Suri \*, Tuomas Sandholm \*\*, and Priyank Ramesh Warkhede \*\*\*

Department of Computer Science

Washington University

St. Louis, MO 63130

{suri,sandholm,priyank}@cs.wustl.edu

**Abstract.** Current IP routers are stateless: they forward individual packets based on the destination address contained in the packet header, but maintain no information about the application or flow to which a packet belongs. This stateless service model works well for best effort datagram delivery, but is grossly inadequate for applications that require quality of service guarantees, such as audio, video, or multimedia. Maintaining state for each flow is expensive because the number of concurrent flows at a router can be in the hundreds of thousands. Thus, stateful solutions such as Intserv (integrated services) have not been adopted for their lack of scalability. Motivated by this dilemma, we formulate and solve the *flow aggregation* problem, where we give an efficient algorithm for computing the smallest set of aggregated flows that encode the forwarding state of individual flows. Such aggregation of state information might increase the viability of Intserv-type protocols.

## 1 Introduction

Current IP networks provide one simple service: the best effort packet delivery, in which no guarantee is made about when or if a packet will be delivered. This simple model allows IP routers to be *stateless*: a router does not need to know anything about the potentially large number of individual connections passing through it; it simply forwards each IP packet based on the destination address contained in the packet header. The routing table entries are highly aggregated—a single entry like 10100\* provides the next hop information for all destinations that start with prefix 10100. When multiple entries match a packet’s destination, the router uses the longest matching prefix rule to forward the packet [3,6,15].

The best-effort service model works well when there is no congestion in the network and the end applications are relatively insensitive to delay (such as file transfer). In reality, parts of the network are frequently and heavily congested,

---

\* Subhash Suri’s research was supported by NSF under grants ANI 9813723 and CCR-9901958.

\*\* Tuomas Sandholm’s research was supported by NSF under CAREER Award IRI-9703122, Grant IRI-9610122, and Grant IIS-9800994.

\*\*\* Priyank Ramesh Warkhede’s research was supported by NSF under grants ANI 9813723 and ANI 9628190.

and a large number of emerging applications are real-time, meaning they are extremely sensitive to delay, such as audio, video, or IP telephony. During network congestion, a router needs to give priority to real-time traffic over non-real-time traffic, and thus adopt a “differentiated services” model. Such a differentiated services model is also attractive to ISPs (Internet Service Providers), who need better traffic management so they can offer different quality services to different customers at different prices.

A differentiated service model can be implemented by maintaining per-flow state in the routers, as proposed by protocols like RSVP in the Intserv model [2], [13]. Stateful routers can provide more powerful and flexible services such as bandwidth allocation, end-to-end latency bounds, protecting well-behaving flows from misbehaving ones, and end-to-end congestion control [9]. Unfortunately, maintaining per-flow state in routers can be prohibitively expensive because the number of flows can be in the hundreds of thousands. Therefore stateful routers do not scale to large sizes as well as the stateless routers. In this paper, we formulate and solve a problem, called *flow aggregation*, which we hope can make stateful routers more scalable. Before we describe flow aggregation, let us briefly explain how current stateless routers forward packets.

In the IP address scheme, each network is assigned a network address, and each host in that network uses the network address as its prefix. (The host address is fixed length, 32 bits, while the network addresses are variable length prefixes.) Each router maintains a routing table, containing a set of address prefixes; associated with each prefix is a “next hop” label. Thus, an entry  $\langle 10100*, A \rangle$  says that a packet whose destination address starts with 10100 should be forwarded to router  $A$ ; the router  $A$  will forward the packet closer to the packet’s ultimate destination. (The symbol ‘\*’ is the wildcard character.)

Routing table entries are highly aggregated—while there are millions of IP hosts, the largest backbone routers have about 50 thousand prefixes [11]. This aggregation has several advantages—smaller table size reduces table memory, improves search time, and it also reduces the routing update traffic. The aggregation does have a cost—to look up a packet’s next hop, we need to find the longest prefix matching the header, which is a more complicated operation than a simple index into a table. For instance, suppose that a router has three prefixes  $0*$ ,  $010*$ , and  $0101*$ , with corresponding next hops  $A$ ,  $B$  and  $C$ . Then, a packet with destination address  $01011$  matches all three but is sent to  $C$ , the longest of the three matches. On the other hand, a packet with address  $01101$  is sent to  $A$ .

## 1.1 Flow-Based Routing

The simple stateless routing, which works well when the network has sufficient capacity and no congestion, is grossly inadequate for real-time applications, such as audio or video, that have stringest delay requirements. *Stateful* routers can implement more sophisticated routing and packet scheduling by using not just the destination address but additional packet header fields and by maintaining information about flows and applications. For instance, an ISP can provide guaranteed quality of service to a company by routing all traffic between two

company sites along a high bandwidth channel, which requires the routers in the ISP network to maintain state information over network address pairs  $(src, dest)$ . In this paper, we will use  $(src, dest)$  pairs to illustrate ideas, though all the results carry over for any header field pair.

A *flow* is defined as a pair  $(src, dest)$ , where  $src$  and  $dest$  are network address prefixes, each at most  $w$  bits long; in IP version 4, these addresses are at most 32 bits. We define a *flow routing entry* to be a tuple  $\langle (src, dest), action \rangle$ , where *action* is the routing action associated with the flow  $(src, dest)$ . The routing action typically is the address of the next hop router to which the packet should be sent, but its exact semantics is irrelevant to our abstract framework; in some applications, the action could also take the form of “do not forward the packet” which is useful for access control [4,5].

We say that a flow routing entry  $(src, dest)$  *matches* a packet  $P$  if  $src$  is a prefix of the packet’s source address, and  $dest$  is a prefix of the packet’s destination address. Thus, a packet with header  $(0011, 1100)$  matches the flow  $(00*, 1*)$ , but not the flow  $(00*, 10*)$ . Let  $\mathcal{D}$  denote a table of  $N$  flow routing entries. Given packet header  $P$ , it is possible that more than one flow entries of  $\mathcal{D}$  match  $P$ , in which case we define the best matching flow, as follows. Suppose two flow entries,  $F_1$  and  $F_2$ , match  $P$ . We say that  $F_1$  is a better match than  $F_2$  if each field of  $P$  has an equal or longer match with  $F_1$  than  $F_2$ . The *best matching flow* of  $P$  is the flow that is a better match than any other matching flow in  $\mathcal{D}$ . For instance, if we consider a packet header  $(0011, 1100)$ , and two flow entries  $F_1 = (001*, 110*)$ , and  $F_2 = (00, 1*)$ . Then,  $F_1$  is the best matching flow for the packet.

In order for the best matching flow to be well-defined, the flow entries must be *consistent*, that is, there cannot be two flow entries that partially overlap in the flow address space. We say that a flow routing table  $\mathcal{D}$  is consistent if for any two flow entries  $F_i$  and  $F_j$  either  $F_i$  and  $F_j$  are disjoint, or one is a subset of the other. Because the primary motivation for flow-based routing is to uniquely classify flows, we will be interested only in consistent flow routing tables. A related work [1] shows how to transform a set of possibly inconsistent classifiers into a consistent ones, by adding additional entries.

## 1.2 Flow Aggregation and Our Contribution

In a consistent flow routing table, each packet header has a unique best matching flow. We say that two flow tables are *equivalent* if each possible packet header receives the same routing action in both tables (using the best matching flow rule). We can define the *flow aggregation* problem as follows: Given a flow routing table  $\mathcal{D}$ , compute another table  $\mathcal{D}'$  that is equivalent to  $\mathcal{D}$  and has the smallest possible number of flow entries. As an example, consider a flow table with the following four entries:  $\langle (00*, 10*), A \rangle$ ,  $\langle (00*, 11*), A \rangle$ ,  $\langle (01*, 10*), A \rangle$ ,  $\langle (01*, 11*), B \rangle$ . The smallest equivalent table for this example has two entries:  $\langle (0*, 1*), A \rangle$ ,  $\langle (01*, 11*), B \rangle$ .

Our main result is a fast algorithm for determining the optimal aggregation. If the input table has  $N$  flow entries, and  $K$  distinct routing actions, and each field

(source or destination) has at most  $w$  bits, then our algorithms runs in worst-case time  $O(NKw^2)$ ; using quadtree style path-compression [12], the worst-case time can be improved to  $O(NK)$ , assuming  $w$  word size.

A pragmatic question one can ask is this: how are flow entries generated, and should one expect any significant aggregation to be achieved? Indeed, if the flow routing entries were manually generated by a network manager, then one would not expect any significant aggregation by running our algorithm. Instead, we expect the flow entries to be generated automatically by various algorithms that are being proposed for dynamic routing and traffic engineering. These protocols can generate a large number of flow entries, and since the number of distinct next hops at each router is much smaller (generally, tens or at most a few hundred in very large backbone routers) than the number of flows, significant aggregation may be achievable. There are also proposals for using packet traces at ISP boundaries to build virtual-circuit paths, such as in multi-protocol label switching (MPLS), which are basically flows routing entries. Like the IP prefix aggregation in stateless routers, flow aggregation has the benefits of improved lookup time and reduced memory. (Reducing memory also leads to improvement in the lookup time, because a smaller data structure may fit entirely in the fast cache [3].)

### 1.3 Previous Work

A lot of work has been done in the networking community on congestion control and end-to-end delay bounds *assuming* that routers maintain flow information [2, 8, 9, 13]. However, we have not seen any algorithmic work on aggregating flow state.

The one-dimensional version of our algorithm solves the flow aggregation problem when flows are defined simply by destination-address prefixes. This turns out to be the *prefix table aggregation* problem, which was solved independently by Daves et al. [7], preceding our work by a few months. The main focus and result in [7] is prefix compaction, while our main motivation and contribution is flow aggregation, which is a two-dimensional problem. We do not believe that the algorithm in [7] generalizes to flow aggregation, and we think our geometric interpretation and resulting dynamic programming are central to solving the flow problem. The flow aggregation problem is formulated as a geometric compression problem in Section 2. We describe the one-dimensional version of our dynamic program in Section 3 primarily to lay the groundwork for the two-dimensional flow aggregation problem. In Section 4 we present our main result: the flow aggregation algorithm. In Section 5 we present some extensions and experimental results. Finally, we conclude in Section 6.

## 2 Flow Entries as Rectangles

We interpret each flow entry as a geometric rectangle in the two-dimensional IP address space—the two axes are the source and the destination addresses.

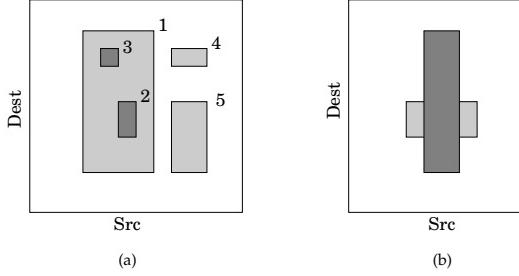


Fig. 1: (a) An example showing 5 consistent rectangles. The number at the top-right corner of each rectangle gives its color (action). A point that lies in both rectangles 1 and 3, receives the color 3, corresponding to the more specific match. A point lying in rectangle 4 gets the color 4. (b) An example of two inconsistent rectangles.

Since each address uses  $w$  bits, the domain is the integer line  $[0, 2^w - 1]$  along each axis. The source and destination fields are network address prefixes, and each such prefix encodes a *contiguous range* of addresses. For instance, the prefix  $101*$  corresponds to the closed interval  $[1010 \dots 0, 1011 \dots 1]$ . The prefix ranges have the property that either two ranges are disjoint, or one contains the other. The range of  $s_1$  contains the range of  $s_2$  precisely when  $s_1$  is a *prefix* of  $s_2$ . For instance, the range of  $10*$  is a superset of the range of  $10110*$ , but the ranges of  $1010*$  and  $110*$  are disjoint. A packet header has fully specified source and destination addresses, and thus corresponds to a *point* in the two-dimensional space.

A flow  $(s, d)$  corresponds to the rectangle whose projections are the ranges of  $s$  and  $d$  in their respective dimensions. We denote this rectangle by  $R(s, d)$ —the points of  $R(s, d)$  are precisely the packet headers that match the flow  $(s, d)$ . To emphasize that we are dealing with special rectangles, we will use the term *prefix rectangle*. We say that two prefix rectangles are *consistent* if they are either disjoint, or one contains the other. The flow table  $\mathcal{D}$  is consistent if all its flow entries are pairwise consistent. Figure 1 shows examples of consistent and inconsistent rectangles.

Consider a flow routing table  $\mathcal{D}$  with  $N$  flow entries. These flows map to  $N$  prefix rectangles in the two-dimensional space  $[0, 2^w - 1] \times [0, 2^w - 1]$ . We let each distinct *action*, associated with our flows, to be represented by a color, where colors are integers numbered from one to  $K$ . Thus, we can think of a flow tuple  $\langle (s, d), \text{action}_i \rangle$  as a prefix rectangle with color  $i$ . Since each packet must be classified into some flow, we assume, without loss of generality, that the prefix rectangles of  $\mathcal{D}$  completely cover the two-dimensional space  $[0, 2^w - 1] \times [0, 2^w - 1]$ . The *flow classification* induced by  $\mathcal{D}$  is the mapping from packet headers (points) to the set of colors. Using the best matching flow rule, each packet header receives a unique color: the color assigned to a point is the color of the *smallest* (most specific) rectangle containing the point. (Refer to Figure 1)

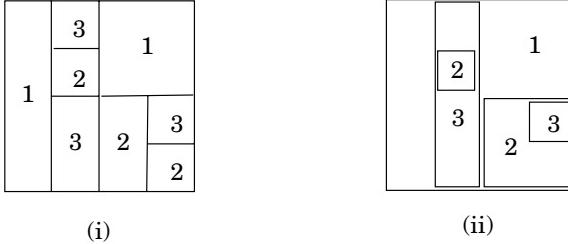


Fig. 2: An example of flow aggregation. Fig. (i) shows an input with 8 rectangles, and Fig. (ii) shows an optimal solution using 5 rectangles.

We can now formulate the flow aggregation problem. Given  $N$  prefix rectangles with colors in  $\{1, 2, \dots, K\}$ , determine the smallest set of consistent prefix rectangles and their colors that induce the same coloring as the input set. Figure 2 shows an example. We begin by considering the problem in one dimension to help us develop the main idea for our algorithm.

### 3 Aggregation in One Dimension

Consider a set of  $N$  prefixes  $\mathcal{D} = \{s_1, s_2, \dots, s_n\}$ , where each  $s_i$  is a binary bit string of length at most  $w$ , and the  $i$ th string is assigned color  $c_i$ , with  $c_i \in \{1, 2, \dots, K\}$ . Each string  $s_i$  corresponds to a contiguous interval on the line  $[0, 2^w - 1]$ , which we call the *prefix range* of  $s_i$ , and denote by  $R(s_i)$ . The set of  $N$  prefix ranges partitions the line  $[0, 2^w - 1]$  into at most  $2N - 1$  “elementary intervals,” where each elementary interval is the interval between two consecutive range endpoints. Assign to each elementary interval the color of the *smallest range* containing that interval. Under this coloring rule, the prefix set  $\mathcal{D}$  is a mapping from the points of the line  $[0, 2^w - 1]$  to the color set  $\{1, 2, \dots, K\}$ . Given a point  $P$ , we let  $\mathcal{D}(P)$  denote the color assigned to  $P$  by the set  $\mathcal{D}$ . Figure 3 shows an example, where a set of prefixes partition the line into six elementary intervals. The colors assigned to these intervals, in left to right order, are 2, 1, 2, 3, 2, 3.

We say that two prefix sets  $\mathcal{D}$  and  $\mathcal{D}'$  are *equivalent* if they induce the same coloring on the line  $[0, 2^w - 1]$ . That is,  $\mathcal{D}(P) = \mathcal{D}'(P)$ , for all  $P \in [0, 2^w - 1]$ . The one-dimensional prefix aggregation problem can be formulated as follows: Given a set of prefixes  $\mathcal{D}$ , find the smallest prefix set  $\mathcal{D}'$  that is equivalent to  $\mathcal{D}$ . Figure 3 (ii) shows the optimal solution for the example in (i); the number next to each prefix range is its color.

Our algorithm uses dynamic programming to compute the optimal set  $\mathcal{D}'$ . We divide a prefix range into two halves, and then try to combine their optimal solutions. One difficulty with this obvious approach is that the combined cost may depend on the actual subproblem solutions. Consider, for instance, the case where we have four equal-length elementary intervals colored 1, 2, 3, 1. The left half subproblem has an optimal solution  $\{1, 2\}$ ; the right half subproblem has



Fig. 3: An example of aggregation in one dimension. Fig. (i) shows an input with 6 prefix ranges, and Fig. (ii) shows an optimal solution using 4 prefix ranges.

an optimal solution  $\{3, 1\}$ . But adding them together does not give the optimal solution, which has only three prefixes. With this motivation, let us introduce the concept of a *background prefix*.

Consider a prefix  $s$ , and its range  $R(s) \subseteq [0, 2^w - 1]$ . Suppose we just want to solve the coloring subproblem for the range  $R(s)$ . We say that a solution  $G$  for  $R(s)$  contains a *background prefix* if  $s \in G$ ; that is, one of the prefix ranges in  $G$  is the whole interval  $R(s)$ . The *background color* of  $G$  is the color of the background prefix. Fig. 3 (ii) shows an example that has a background prefix with color 2, while the set of prefixes in Fig. 3 (i) does not contain a background prefix. Our dynamic programming algorithm will use the key observation that it is sufficient to consider solutions in which background colors are well-defined.

**Lemma 1.** *Every solution of the coloring problem for a prefix range  $R(s)$  can be modified into a solution of equal cost with a background prefix.*

*Proof.* Consider a solution without a background prefix. Pick a prefix  $p$  in this solution such that the range  $R(p)$  is not contained in any other prefix's range. Replace  $p$  by  $s$ , and give it  $p$ 's color.

### 3.1 The Dynamic Programming Algorithm

We are given a set  $\mathcal{D} = \{s_1, s_2, \dots, s_n\}$  of  $N$  prefixes, where each  $s_i$  is a binary bit string of maximum length  $w$ , and the  $i$ th string is assigned color  $c_i$ , with  $c_i \in \{1, 2, \dots, K\}$ . Consider the coloring induced by  $\mathcal{D}$  on the line  $[0, 2^w - 1]$ : a point has the color of the smallest prefix range in which it lies. (Note that fewer the bits in a prefix  $s_i$ , the longer the corresponding range  $R(s_i)$  is. The null string  $*$  corresponds to the whole range  $[0, 2^w - 1]$ , while a full  $w$ -bit string maps to a point.) We start by building a partition of  $[0, 2^w - 1]$  in which each piece is monochromatic and each interval has length a binary power. That is, we recursively divide the line  $[0, 2^w - 1]$  into two equal halves until each piece is monochromatic. Because  $\mathcal{D}$  has  $N$  prefixes, and each prefix has at most  $w$  bits, our final subdivision has size at most  $wN$ .

Let  $p_1, p_2, \dots, p_M$ , where  $M \leq wN$ , denote the prefixes that correspond to the monochromatic intervals in the final subdivision. We call the  $R(p_i)$ 's *monochromatic binary* intervals. These intervals are the basic subproblems for our dynamic program's initialization. Given an arbitrary prefix range  $R(s) \subseteq [0, 2^w - 1]$ , and a color  $c \in \{1, 2, \dots, K\}$ , let us define

$\text{cost}(s, c) =$  value of an optimal solution for the range  $R(s)$  with background color  $c$ .

We initialize this cost function for the monochromatic binary intervals  $R(p_i)$ , as follows. Let  $c_0(p_i)$  be the color of the interval  $R(p_i)$ —that is,  $c_0(p_i)$  is the color induced on  $R(p_i)$  by the input prefix set  $\mathcal{D}$ . Then, for  $i = 1, 2, \dots, M$ , we let  $\text{cost}(p_i, c) = 1$  if  $c = c_0(p_i)$ , and  $\text{cost}(p_i, c) = \infty$  otherwise. The following lemma gives the general formula for this cost function. Given a prefix  $s$ , we use  $s0$  and  $s1$  to denote strings obtained by appending to  $s$  a 0 and a 1, respectively.

**Lemma 2.** *Let  $s$  be an arbitrary prefix. Then, for  $c_i = 1, 2, \dots, K$ ,*

$$\text{cost}(s, c_i) = \min \begin{cases} \text{cost}(s0, c_i) + \text{cost}(s1, c_i) - 1 & c_i \neq c_j \\ \text{cost}(s0, c_i) + \text{cost}(s1, c_j) & c_i \neq c_j \\ \text{cost}(s0, c_j) + \text{cost}(s1, c_i) & c_i \neq c_j \\ \text{cost}(s0, c_j) + \text{cost}(s1, c_l) + 1 & c_i \neq c_j, c_l \end{cases}$$

*Proof.* Omitted from this extended abstract for lack of space.

If the input  $\mathcal{D}$  has  $N$  prefixes, the number of colors is  $K$ , and the prefixes are  $w$  bits long, then the dynamic program based on Lemma 2 takes  $O(NKw)$  time and space. When we implemented our algorithm, we found that the worst-case memory requirement for this algorithm was infeasibly large to be of practical value. For instance, for the practical values of interest  $N = 50,000$ ,  $w = 32$ , and  $K = 256$ , the dynamic program needs to construct a table of size  $4 \times 10^8$ . Even assuming that each entry takes just 1 word of memory, the worst-case memory requirement for this algorithm is 3200 MB of memory! This motivated us to look for an improved algorithm, which we describe in the next section. Not only does the new algorithm require significantly less memory in practice, but it is also simpler and faster.

### 3.2 An Improved Dynamic Program

Intuitively, maintaining  $K$  distinct solutions, one for each background color, for every subproblem seems like an overkill. (If we were only interested in the *value* of the solution then, we could of course choose not to store the intermediate solutions. However, they are needed for constructing the optimal prefix set.) But as we saw earlier, keeping just one optimal solution does not work. However, we show below that storing just the background colors that give the smallest cost for each subproblem suffices. Let  $s$  be an arbitrary prefix, and let  $\mathcal{L}(s)$  be the list of background colors that give the minimum cost solutions for  $R(s)$ . That is,

$$\mathcal{L}(s) = \{c_i \mid \text{cost}(s, c_i) \leq \text{cost}(s, c_j), \quad 1 \leq i, j \leq k\}$$

Again, we initialize these lists for the monochromatic binary intervals by setting  $\mathcal{L}(p) = \{c_0(p)\}$ . The following lemma shows how to compute these lists in a bottom-up merge. (Recall that  $s0$  and  $s1$  are prefixes obtained by appending 0 and 1 to the prefix  $s$ .)

**Lemma 3.** Suppose  $s$  is an arbitrary prefix. Then,

$$\mathcal{L}(s) = \begin{cases} \mathcal{L}(s0) \cap \mathcal{L}(s1) & \text{if } \mathcal{L}(s0) \cap \mathcal{L}(s1) \neq \emptyset \\ \mathcal{L}(s0) \cup \mathcal{L}(s1) & \text{otherwise.} \end{cases}$$

*Proof.* We first consider the case  $\mathcal{L}(s0) \cap \mathcal{L}(s1) \neq \emptyset$ . Since all colors in the intersection set  $\mathcal{L}(s0) \cap \mathcal{L}(s1)$  are equivalent, it would suffice to show that  $\text{cost}(s, c_i) < \text{cost}(s, \bar{c})$  whenever  $c_i \in \mathcal{L}(s0) \cap \mathcal{L}(s1)$  and  $\bar{c} \notin \mathcal{L}(s0) \cap \mathcal{L}(s1)$ . It is easy to see that

$$\text{cost}(s, c_i) = \text{cost}(s0, c_i) + \text{cost}(s1, c_i) - 1.$$

(That is, the minimum is achieved by the first term in the expression of Lemma 2.) Assume, without loss of generality, that  $\bar{c} \notin \mathcal{L}(s0)$ . Then we must have  $\text{cost}(s0, \bar{c}) \geq 1 + \text{cost}(s0, c_i)$ , and  $\text{cost}(s1, \bar{c}) \geq \text{cost}(s1, c_i)$ ; if  $\bar{c} \notin \mathcal{L}(s1)$ , the second inequality is strict. Now, it is easy to check that

$$\text{cost}(s, \bar{c}) = \min \begin{cases} \text{cost}(s0, \bar{c}) + \text{cost}(s1, \bar{c}) - 1 \\ \text{cost}(s0, c_i) + \text{cost}(s1, \bar{c}) \end{cases}$$

Since  $\text{cost}(s0, \bar{c}) \geq 1 + \text{cost}(s0, c_i)$ , and  $\text{cost}(s1, \bar{c}) \geq \text{cost}(s1, c_i)$ , it follows that  $\text{cost}(s, \bar{c}) \geq \text{cost}(s0, c_i) + \text{cost}(s1, c_i) > \text{cost}(s, c_i)$ , which proves the claim.

Next, consider the case  $\mathcal{L}(s0) \cap \mathcal{L}(s1) = \emptyset$ . In this case we show that  $\text{cost}(s, c_i) < \text{cost}(s, \bar{c})$  whenever  $c_i \in \mathcal{L}(s0) \cup \mathcal{L}(s1)$  and  $\bar{c} \notin \mathcal{L}(s0) \cup \mathcal{L}(s1)$ . Let us assume that  $c_i \in \mathcal{L}(s0)$ , and thus  $c_i \notin \mathcal{L}(s1)$ . Then,

$$\text{cost}(s, c_i) = \text{cost}(s0, c_i) + \text{cost}(s1, c_j),$$

for any  $c_j \in \mathcal{L}(s1)$ . Now, since  $\bar{c} \notin \mathcal{L}(s0) \cup \mathcal{L}(s1)$ , we have  $\text{cost}(s0, \bar{c}) \geq 1 + \text{cost}(s0, c_i)$ , and  $\text{cost}(s1, \bar{c}) \geq 1 + \text{cost}(s1, c_j)$ . Since

$$\text{cost}(s, \bar{c}) = \min \begin{cases} \text{cost}(s0, \bar{c}) + \text{cost}(s1, \bar{c}) - 1 \\ \text{cost}(s0, \bar{c}) + \text{cost}(s1, c_j) \\ \text{cost}(s0, c_i) + \text{cost}(s1, \bar{c}), \end{cases}$$

it follows that  $\text{cost}(s, \bar{c}) \geq 1 + \text{cost}(s0, c_i) + \text{cost}(s1, c_j) > \text{cost}(s, c_i)$ , which completes the proof.

Lemma 3 gives a straightforward dynamic programming algorithm. Starting from the initial color lists of the monochromatic binary intervals, the algorithm computes the lists for increasing longer prefix ranges. When computing the list for prefix  $s$ , we set  $\mathcal{L}(s) = \mathcal{L}(s0) \cap \mathcal{L}(s1)$  if  $\mathcal{L}(s0) \cap \mathcal{L}(s1) \neq \emptyset$ ; otherwise  $\mathcal{L}(s) = \mathcal{L}(s0) \cup \mathcal{L}(s1)$ . Once all the lists have been computed, we can determine an optimal color assignment by a top-down traversal. (Details are presented in the full paper.)

The worst-case complexity of the preceding algorithm is  $O(NKw)$ , since there are  $O(Nw)$  subproblems, and the size of a color list is at most  $K$ . Thus,

from a *worst-case* point of view, the dynamic program based on Lemma 3 is not much better than that of Lemma 2. However, in practice we found that the list sizes were much smaller than the total number of colors, and thus the memory requirement was substantially improved. We next describe our main result: the dynamic programming algorithm for the flow aggregation.

## 4 Optimal Flow Aggregation

Consider a set  $\mathcal{D}$  of  $N$  consistent flows. Each flow  $(s, d)$  corresponds to a rectangle  $R(s, d)$  in the two-dimensional space  $[0, 2^w - 1] \times [0, 2^w - 1]$ . The color of  $R(s, d)$  is the color (action) associated with flow  $(s, d)$ . Using the best matching flow rule, the set  $\mathcal{D}$  gives a mapping from the set of points  $[0, 2^w - 1] \times [0, 2^w - 1]$  to the set of colors. Let  $\mathcal{D}(P)$  denote the color assigned to point  $P$  by  $\mathcal{D}$ . Geometrically,  $\mathcal{D}(P)$  is the color of the smallest rectangle containing  $P$ . Our goal is to find the smallest set of consistent flows  $\mathcal{D}'$  that realizes the same coloring map as  $\mathcal{D}$ ; that is,  $\mathcal{D}(P) = \mathcal{D}'(P)$  for all points  $P$ . Our algorithm generalizes the dynamic program of the preceding section.

We start with the observation that any solution can be modified to contain a background flow. The *background flow* for a prefix rectangle  $R(s, d)$  is the flow  $(s, d)$ . We say that a solution  $G$  for the rectangle  $R(s, d)$  contains the background flow if  $(s, d) \in G$ . The background color of  $G$  is the color assigned to the flow  $(s, d)$ . Fig. 2(ii) shows an example that has a background flow of color 1; the set of flows in Fig. 2(i) does not contain a background flow. The following generalizes the background prefix lemma; we omit its easy proof in this abstract.

**Lemma 4.** *Every solution of the coloring problem for a prefix rectangle  $R(s, d)$  can be modified into a solution of equal cost with a background flow.*

Given a prefix rectangle  $R(s, d)$ , and a color  $c \in \{1, 2, \dots, K\}$ , define

$\text{cost}(s, d, c) =$  value of an optimal solution for rectangle  $R(s, d)$  with background color  $c$ .

The following lemma gives the general formula for this cost function. (Recall that we use the notation  $x0$  (resp.  $x1$ ) to denote the bit string  $x$  with 0 (resp. 1) appended.)

**Lemma 5.** *Given a prefix rectangle  $R(s, d)$ , and a color  $c_i \in \{1, 2, \dots, K\}$ , we have*

$$\text{cost}(s, d, c_i) = \min \begin{cases} \text{cost}(s0, d, c_i) + \text{cost}(s1, d, c_i) - 1 & \\ \text{cost}(s0, d, c_i) + \text{cost}(s1, d, c_j) & c_j \neq c_i \\ \text{cost}(s0, d, c_j) + \text{cost}(s1, d, c_i) & c_j \neq c_i \\ \text{cost}(s, d0, c_i) + \text{cost}(s, d1, c_i) - 1 & \\ \text{cost}(s, d0, c_i) + \text{cost}(s, d1, c_j) & c_j \neq c_i \\ \text{cost}(s, d0, c_j) + \text{cost}(s, d1, c_i) & c_j \neq c_i \end{cases}$$

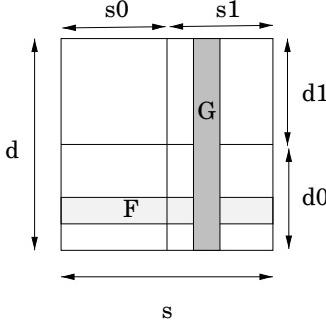


Fig. 4:  $F$  spans  $R(s, d)$  along the  $s$ -axis;  $G$  spans it along the  $d$ -axis.

*Proof.* Omitted due to lack of space.

The key insight in the preceding dynamic program is the following geometric fact: since we are computing consistent rectangles, an optimal solution cannot have two prefix rectangles that cross each other. Thus, an optimal solution for  $R(s, d)$  with background color  $c$  must be composed of either the optimal solutions of the left and right half subproblems, or the top and bottom half subproblems. More specifically, let us introduce the following definition.

We say that a prefix rectangle  $R' = (s', d')$  spans  $R(s, d)$  along the  $s$ -axis (resp.  $d$ -axis) if  $s = s'$  and  $d$  is a prefix of  $d'$  (resp.  $d = d'$  and  $s$  is a prefix of  $s'$ ). Figure 4 illustrates this definition. Consistency implies that an optimal solution of  $R(s, d)$ , with any background color, cannot have rectangles spanning  $R(s, d)$  along both axes. Absence of a rectangle spanning along  $s$ -axis (resp.  $d$ -axis) allows combining left and right (resp. top and bottom) subproblem solutions.

#### 4.1 An Improved Algorithm

As in the one-dimensional case, the dynamic program can be improved in practice (though not in the worst case) by maintaining the list of only those background colors that give optimal solutions. Let  $\mathcal{L}(s, d)$  denote the list of colors that achieve minimum cost for the coloring subproblem  $R(s, d)$ . That is,

$$\mathcal{L}(s, d) = \{c_i \mid \text{cost}(s, d, c_i) \leq \text{cost}(s, d, c_j), 1 \leq c_i, c_j \leq K\}$$

We use the notation  $\text{cost}(s, d)$  to denote the minimum cost of  $R(s, d)$  over all colors; that is,  $\text{cost}(s, d) = \min_i \text{cost}(s, d, c_i)$ . In the following, we use the term “input rectangle” to mean the prefix rectangle corresponding to a flow in the input set  $\mathcal{D}$ .

#### Flow-Aggregate ( $s, d$ )

1. If no input rectangle of  $\mathcal{D}$  lies entirely inside  $R(s, d)$ , then all points mapped to the region  $R(s, d)$  receive the same color  $c$ . In this case, we set  $\mathcal{L}(s, d) = \{c\}$ ,  $\text{cost}(s, d) = 1$ , and **return**.

2. If no input rectangle of  $\mathcal{D}$  spans  $R(s, d)$  along the  $s$ -axis, then do the following:
  - if**  $\mathcal{L}(s0, d) \cap \mathcal{L}(s1, d) \neq \emptyset$  **then**  
 $\text{cost}_v(s, d) = \text{cost}(s0, d) + \text{cost}(s1, d) - 1;$      $\mathcal{L}_v(s, d) = \mathcal{L}(s0, d) \cap \mathcal{L}(s1, d)$   
**else**  $\text{cost}_v(s, d) = \text{cost}(s0, d) + \text{cost}(s1, d);$      $\mathcal{L}_v(s, d) = \mathcal{L}(s0, d) \cup \mathcal{L}(s1, d)$   
**Otherwise**, set  $\text{cost}_v(s, d) = \infty.$
3. If no input rectangle of  $\mathcal{D}$  spans  $R(s, d)$  along the  $d$ -axis, then do the following:
  - if**  $\mathcal{L}(s, d0) \cap \mathcal{L}(s, d1) \neq \emptyset$  **then**  
 $\text{cost}_h(s, d) = \text{cost}(s, d0) + \text{cost}(s, d1) - 1;$      $\mathcal{L}_h(s, d) = \mathcal{L}(s, d0) \cap \mathcal{L}(s, d1)$   
**else**  $\text{cost}_h(s, d) = \text{cost}(s, d0) + \text{cost}(s, d1);$      $\mathcal{L}_h(s, d) = \mathcal{L}(s, d0) \cup \mathcal{L}(s, d1)$   
**Otherwise**, set  $\text{cost}_h(s, d) = \infty.$
4. **if**  $\text{cost}_v(s, d) > \text{cost}_h(s, d)$  **then**  
 $\text{cost}(s, d) = \text{cost}_h(s, d);$      $\mathcal{L}(s, d) = \mathcal{L}_h(s, d)$   
**else if**  $\text{cost}_v(s, d) < \text{cost}_h(s, d)$  **then**  
 $\text{cost}(s, d) = \text{cost}_v(s, d);$      $\mathcal{L}(s, d) = \mathcal{L}_v(s, d)$   
**else**  $\text{cost}(s, d) = \text{cost}_v(s, d);$      $\mathcal{L}(s, d) = \mathcal{L}_h(s, d) \cup \mathcal{L}_v(s, d)$

The code above describes a generic call on an arbitrary prefix rectangle  $R(s, d)$ . The initial call is made on the subspace  $R(*, *)$ , corresponding to the rectangle  $[0, 2^w - 1] \times [0, 2^w - 1]$ . In the code,  $\text{cost}_h$ ,  $\text{cost}_v$ ,  $\mathcal{L}_h$  and  $\mathcal{L}_v$  are temporary variables used for comparing the solutions obtained by either combining the left and right halves of  $R(s, d)$ , or the top and bottom halves. Due to lack of space, we omit the proof of correctness of this algorithm.

In order to analyze the running time of this dynamic program, we observe that a subproblem  $R(s, d)$  makes a recursive call only if  $R(s, d)$  contains at least one input rectangle of  $\mathcal{D}$  inside it. We can show that the total number of subproblems is  $O(Nw)$ , the cost of deciding if a rectangular region is spanned by some filter is  $O(w)$ , and the cost of maintaining color lists per subproblem is  $O(K)$ . Thus, the total time and space complexity of the algorithm is  $O(NKw^2)$  in the worst case.

**Theorem 1.** *Given a set of  $N$  consistent flows, with  $K$  distinct colors and at most  $w$ -bit prefixes, we can compute an optimal flow aggregation in  $O(NKw^2)$  worst case time.*

## 5 Extensions and Experimental Results

### 5.1 Improving Time Complexity by Path Compression

The dynamic programs of Sections 3 and 4 can be improved to eliminate the  $w$  factors, thus resulting in the worst-case running time and space  $O(NK)$ . The  $w$  factors arise due to long non-branching paths in the recursion tree. A standard quadtree style path compression can eliminate such paths, by shrinking the rectangle  $R(s, d)$  in each step to ensure that each recursive call separates two input rectangles.

Table 1: One dimensional prefix aggregation. When multiple next hops were available for a prefix, we initialized the corresponding color list with all those next hops. The input and output are the number of prefixes.

Database	Input	Output	Reduction	Memory	Time
Mae-East	41455	23680	42.88%	3.8 MB	2.73 s
PacBell	24728	14168	42.70%	2.1 MB	1.85 s
Paix	7982	5888	26.23%	0.8 MB	0.72 s

**Minimizing the Bit Complexity** We have used the number of flows as our complexity measure. Instead one could ask to minimize the total *bit complexity* of the flow routing table. Algorithms that use tries or bit vectors for flow classification [3, 8, 10, 14] are sensitive to the total number of bits in the routing database. Given a flow  $f = (s, d)$ , let  $b(f)$  denote the bit length of  $s$  plus the bit length of  $d$ . Then, the *bit complexity* of a flow routing table  $\mathcal{D} = \{f_1, f_2, \dots, f_n\}$  is  $\sum_{i=1}^n b(f_i)$ . We could ask for a routing table of minimum bit complexity that is equivalent to  $\mathcal{D}$ . It turns out that our dynamic programs also minimizes the bit complexity of the output table.

## 5.2 Experimental Results

We implemented our dynamic programming algorithms, for both one- and two-dimensional aggregation. We do not have any publically available flow databases to test our two-dimensional algorithm, since the stateful routers are still in their infancy. On the other hand, prefix tables are widely available for large backbone routers, so we were able to test our one-dimensional aggregation algorithm. We ran our algorithm on three publically available routing tables, obtained from the Mae-East Exchange Point [11]. The number of prefixes in these databases varied from about 8000 (Paix) to about 41000 (Mae-East). The total number of colors (distinct next hops) varied from 17 to 58. Table 1 below shows our results. While one-dimensional results are no indication of the two dimensional problem, it should be encouraging that our prefix aggregation algorithm achieves compression of 30-40% even in these highly aggregated prefix tables. It therefore appears likely that significant aggregation might be possible in the flow routing tables, which are going to be automatically generated.

## 6 Concluding Remarks

We gave an efficient algorithm for computing an optimal flow aggregation for reducing state information in IP routers. The algorithm is relatively simple, and exploits some basic geometric properties of consistent prefix rectangles in two dimensions. The basic dynamic programming algorithm runs in  $O(NKw^2)$  worst-case time, for  $N$  flows with  $K$  colors and  $w$  bit prefixes. While the improved dynamic program does not reduce the worst-case complexity, it should

be substantially better in practice. With path compression in the recursion tree, the worst-case time can be reduced to  $O(NK)$ .

The IP routers certainly need to move beyond the current best-effort service model, if they are to be used for advanced services like audio, video, or IP telephony. The past history has shown that highly stateful solutions like ATM (asynchronous transfer mode) have failed to be widely adopted despite their many ability to provide quality of service. Achieving similar capabilities in IP routers with minimal per-flow state appears to be the most promising alternative. Our hope is that algorithms like ours for flow aggregation will make stateful routers more scalable, and thus more acceptable.

## References

1. H. Adiseshu, S. Suri, and G. Parulkar. Detecting and resolving packet filter conflicts. In *Proc. of IEEE INFOCOM*, 2000.
2. R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource reservation protocol (RSVP)—Version 1, functional specification. RFC 2205, September 1997.
3. A. Brodnik, S. Carlsson, M. Degermark, and S. Pink. Small forwarding table for fast routing lookups. In *Proc. of ACM SIGCOMM*, 1997.
4. D.B. Chapman and E.D. Zwicky. *Building Internet Firewalls*. O'Reilly & Associates, Inc., 1995.
5. W. Cheswick and S. Bellovin. *Firewalls and Internet Security*. Addison-Wesley, 1995.
6. G. Cheung and S. McCanne. Optimal routing table design for IP address lookups under memory constraints. In *Proc. of IEEE INFOCOM*, 1999.
7. R. Daves, C. King, S. Venkatachary, and B. Zill. Constructing optimal IP routing tables. In *Proc. of IEEE INFOCOM*, 1999.
8. D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: A software architecture for next generation routers. In *Proc. of ACM SIGCOMM*, 1998.
9. A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *Journal of Internetworking Research and Experience*, pages 3–26, 1990.
10. T.V. Lakshman and D. Stidialis. High speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proc. of ACM SIGCOMM*, 1998.
11. Inc. Merit. Routing table snapshot, 14 Jan '99, Mae-East NAP. <ftp://ftp.merit.edu/statistics/ipma>.
12. J. Mitchell, D. Mount, and S. Suri. Query-sensitive ray shooting. *Int. Journal of Computational Geometry & Applications*, pages 317–347, 1997.
13. S. Shenker, R. Braden, and D. Clark. Integrated services in the Internet architecture: an overview. RFC 1633, June 1994.
14. V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel. Fast scalable level four switching. In *Proc. of ACM SIGCOMM*, 1998.
15. M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. In *Proc. of ACM SIGCOMM*, 1997.

# On the Complexities of the Optimal Rounding Problems of Sequences and Matrices

Tetsuo Asano<sup>1</sup>, Tomomi Matsui<sup>2</sup>, and Takeshi Tokuyama<sup>3</sup>

<sup>1</sup> School of Information Science, Japan Advanced Institute of Science and Technology, [t-asano@jaist.ac.jp](mailto:t-asano@jaist.ac.jp)

<sup>2</sup> Department of Information Engineering, University of Tokyo, [tomomi@misojiro.t.u-tokyo.ac.jp](mailto:tomomi@misojiro.t.u-tokyo.ac.jp)

<sup>3</sup> Graduate School of Information Sciences, Tohoku University, [tokuyama@dais.is.tohoku.ac.jp](mailto:tokuyama@dais.is.tohoku.ac.jp)

**Abstract.** In this paper, we discuss the problem of computing an optimal rounding of a real sequence (resp. matrix) into an integral sequence (resp. matrix). Our criterion of the optimality is to minimize the weighted  $l_\infty$  distance  $Dist_\infty^{\mathcal{F}, w}(A, B)$  between an input sequence (resp. matrix)  $A$  and the output  $B$ . The distance is dependent on a family  $\mathcal{F}$  of intervals (resp. rectangular regions) for the sequence rounding (resp. matrix rounding) and positive valued weight function  $w$  on the family. We give efficient polynomial time algorithms for the sequence-rounding problem, one for the weighted  $l_\infty$  distance, and the other for any weight function  $w$ , for any family  $\mathcal{F}$  of intervals. We give an algorithm that computes a matrix rounding with an error at most 1.75 with respect to the unweighted  $l_\infty$  distance associated with the family  $\mathcal{W}_2$  of all  $2 \times 2$  square regions, whereas we prove that it is NP-hard to compute an approximate solution to the matrix-rounding problem with an approximate ratio smaller than 2 for the same distance.

## 1 Introduction

Given a real number  $\alpha$ , its *rounding* is either  $\lfloor \alpha \rfloor$  or  $\lceil \alpha \rceil$ . Given a  $d$ -dimensional  $n \times n \times \cdots \times n$  array ( $n^d$  array)  $A = (a_{i_1, i_2, \dots, i_d})_{1 \leq i_j \leq n}$  of real numbers, its rounding is an integral  $n^d$  array  $B = (b_{i_1, i_2, \dots, i_d})_{1 \leq i_j \leq n}$  such that each entry  $b_{i_1, i_2, \dots, i_d}$  is a rounding of  $a_{i_1, i_2, \dots, i_d}$ . Without loss of generality, we assume that each entry of  $A$  is in the closed interval  $[0, 1]$ . Such an array is called a  $[0, 1]$ -valued  $n^d$  array. Thus, a rounding of  $A$  becomes a binary array.

Given an array  $A$ , there are  $2^{n^d}$  possible roundings, among which a “good-quality” rounding is desired. In order to give a criterion to evaluate the quality of a rounding, we define a distance in the space  $\mathcal{A}$  of all  $[0, 1]$ -valued  $n^d$  arrays. An *orthogonal region*  $R$  in the  $d$ -dimensional integral grid  $[1, n]^d$  is a Cartesian product  $I_1 \times I_2 \times \cdots \times I_d$  of integral subintervals  $I_j$  of  $[1, n]$  for  $j = 1, 2, \dots, d$ . For an element  $A \in \mathcal{A}$ , let  $A(R)$  be the sum of entries of  $A$  located in the orthogonal region  $R$ . Given a family  $\mathcal{F}$  of orthogonal regions, the associated  $l_p$  distance

$Dist_p^{\mathcal{F}}(A, A')$  between two elements  $A$  and  $A'$  in  $\mathcal{A}$  is defined by

$$Dist_p^{\mathcal{F}}(A, A') = \left[ \sum_{R \in \mathcal{F}} |A(R) - A'(R)|^p \right]^{1/p}.$$

The  $l_\infty$  distance with respect to  $\mathcal{F}$  is defined by

$$Dist_{\infty}^{\mathcal{F}}(A, A') = \lim_{p \rightarrow \infty} Dist_p^{\mathcal{F}}(A, A') = \max_{R \in \mathcal{F}} |A(R) - A'(R)|.$$

More generally, we could consider a positive valued function  $w$  on  $\mathcal{F}$  and define the *weighted  $l_\infty$  distance*

$$Dist_{\infty}^{\mathcal{F}, w}(A, A') = \max_{R \in \mathcal{F}} |(A(R) - A'(R))w(R)|.$$

Let  $\mathcal{B}$  be the set of all binary arrays in  $\mathcal{A}$ . Given a  $[0, 1]$ -valued  $n^d$  array  $A$ , an *optimal rounding* of  $A$  is a binary  $n^d$  array  $B$  in  $\mathcal{B}$  that is closest to  $A$  in the sense of the above-defined distance. The distance between  $A$  and its optimal rounding is referred to as the *optimal rounding error*. In this paper, we are mainly concerned with the weighted and unweighted  $l_\infty$  distances.

The supremum  $\sup_{A \in \mathcal{A}} \min_{B \in \mathcal{B}} Dist_{\infty}^{\mathcal{F}}(A, B)$  of the optimal rounding error is called the *inhomogeneous discrepancy* of  $\mathcal{A}$  with respect to the family  $\mathcal{F}$  or with respect to the distance  $Dist_{\infty}^{\mathcal{F}}$ . (See Beck and SöS [10]). We deal with the following problems:

Problem 1 (Discrepancy problem). Give combinatorial upper and lower bounds of the inhomogeneous discrepancy of  $\mathcal{A}$  with respect to  $Dist_{\infty}^{\mathcal{F}}$ .

Problem 2 (Optimization problem). How computationally hard is the problem of computing an optimal rounding of a  $[0, 1]$ -valued  $n^d$  array?

The discrepancy problem is a classical topic in combinatorics, and our main focus is on the optimization problem. In particular, we consider two special cases where  $d = 1$  and  $d = 2$ , which are called the *sequence-rounding problem* and the *matrix-rounding problem*, respectively. These problems are not only combinatorially interesting but also related to coding theory, data compression, computer vision, operations research, and Monte Carlo simulation.

For the sequence-rounding problem, the inhomogeneous discrepancy with respect to  $Dist_{\infty}^{\mathcal{F}}$  is at most 1 for any family  $\mathcal{F}$  of intervals. On the other hand, it can be 1 even if we consider the family of all intervals of length 2. Therefore, the discrepancy problem is easily settled for the sequence-rounding. On the other hand, to the authors' knowledge, the optimization problem has not been addressed well in the literature. Viterbi [21] considered the  $l_1$  distance with respect to the family of all intervals of length  $k$  in application to a decoding problem, and proposed an  $O(2^k n)$  time algorithm. Although a similar algorithm could be applied to the  $l_\infty$  distance, the time complexity would be exponential in general. We show in this paper that an optimal rounding of any sequence with respect to any weighted  $l_\infty$  distance can be computed in  $O(\sqrt{n}|\mathcal{F}| \log^2 n)$  time. The time

complexity is polynomial since  $|\mathcal{F}| = O(n^2)$ . We also give an  $O(k^2 n \log n)$  time algorithm if the maximal length of the intervals of  $\mathcal{F}$  is  $k$ .

For the matrix-rounding problem, the inhomogeneous discrepancy with respect to  $Dist_{\infty}^{\mathcal{F}}$  highly depends on the choice of the family  $\mathcal{F}$  of regions: If  $\mathcal{F}$  is the set of all orthogonal regions, an  $O(\log^3 n)$  upper bound and an  $\Omega(\log n)$  lower bound are known [10]. On the other hand, Baranyai [7] showed that the inhomogeneous discrepancy is less than 1 if  $\mathcal{F}$  consists of  $2n + 1$  regions corresponding to all rows, all columns and the whole matrix (see [28][10] for its applications). Moreover, it is known that the inhomogeneous discrepancy is less than 2 for a set  $\mathcal{F}$  consisting of intervals in any two different scanning orderings on the entries of the matrix [10].

In this paper, motivated from an application to *digital halftoning*, we would like to consider the family  $\mathcal{W}_k$  consisting of all  $k \times k$  square regions for a small  $k$ . An  $O(\log^3 k)$  upper bound and an  $\Omega(\log k)$  lower bound of the inhomogeneous discrepancy can be easily obtained from the above mentioned known results. Our main results for the matrix-rounding are on the family  $\mathcal{W}_2$ . We give a nontrivial 1.75 upper bound for the inhomogeneous discrepancy  $\sup_{A \in \mathcal{A}} \min_{B \in \mathcal{B}} Dist_{\infty}^{\mathcal{W}_2}(A, B)$ , whereas we prove that it is NP-hard to approximate the rounding error with respect to  $Dist_{\infty}^{\mathcal{W}_2}$  within the factor 2. The NP-hardness result is generalized for  $\mathcal{W}_{2k}$  for any natural number  $k$ .

Our motivation comes from digital halftoning, which is one of the most fundamental techniques in image processing. An intensity image can be considered as a  $[0, 1]$ -valued  $n \times n$  array  $A$  where each entry  $a_{i,j}$  corresponds to a brightness level (gray level) of the  $(i, j)$  pixel of the pixel grid. Its digital halftoning is a binary  $n \times n$  array  $B$  “approximating”  $A$ . The intention of this method is to convert a given image which consists of several bits for brightness levels into a binary image having only black and white pixels. This kind of technique is indispensable to print an image on an output device that produces black dots only, such as facsimiles and laser printers.

Up to now, a large number of algorithms for digital halftoning have been proposed (see, e.g., [16][13][6][17][18]). A comprehensive summary of the results obtained in the literature can be found in the Ph. D. Thesis by Ulichney [20]. However, there have been few studies discussing reasonable criteria for evaluating the quality of an output image; maybe because the problem itself is very practically oriented. Actually, the most common criterion on digital halftoning is to judge the quality of output pictures by human eyes. It is desirable to establish a good evaluation system of halftoning methods (instead of the “human eye’s judgment”), and to handle the digital halftoning problem fully mathematically. The idea of using discrepancy for measuring the smoothness of halftoning has been given [10][19]; however, to the authors’ knowledge, the discrepancy with respect to families of small regions and its computational aspect has not been well studied before.

Imagine that we look at some pixel  $(i, j)$  of a gray-level image  $A$ . What happens is, we actually perceive an average of gray levels of some small neighborhood of that point. Using the same observation, the intensity around the

pixel  $(i, j)$  of a binary image is proportional to the number of white points in the corresponding neighborhood. Therefore, density values should be roughly equal around *any* pixel between an output binary image and the input image  $A$ . The observation motivates us to consider the family  $\mathcal{W}_k$  for a small  $k$ . Indeed, a weighted  $l_\infty$  distance for the family  $\cup_{i=1}^k \mathcal{W}_i$  seems to be a nice criterion for the digital halftoning problem. If an optimal rounding with respect to the above mentioned distance were computed in polynomial time, we could have designed an ideal automatic digital halftoning system with a concrete mathematical criterion. Unfortunately, our NP-hardness result implies that we need heuristics to solve the digital halftoning problem formulated as a matrix-rounding problem. One popular heuristic approach is to transform the digital halftoning problem into a one-dimensional problem by using a space-filling curve generated in somewhat random manner [4], where we can apply our sequence-rounding algorithm to solve the one-dimensional problem.

## 2 Sequence-Rounding Problem

### 2.1 Supremum of the Optimal Rounding Error

Let  $\mathbf{a} = (a_1, a_2, \dots, a_n)$  be our input sequence such that  $0 \leq a_j \leq 1$  for all  $j \in \{1, 2, \dots, n\}$ .

A popular algorithm used in digital halftoning to round such a sequence  $\mathbf{a}$  is the *error diffusion algorithm*, which computes the binary sequence  $\mathbf{b}$  from  $b_1$  to  $b_n$  greedily in an incremental fashion. We always keep the difference  $S_j = \sum_{i=1}^j (a_i - b_i)$ . If we have already computed  $b_1$  through  $b_j$ , we determine  $b_{j+1}$  to be 1 if  $S_j + a_{j+1} > 0.5$  and to be 0 otherwise. It can be easily seen that  $-0.5 < S_j \leq 0.5$  always holds, and hence for any interval  $I = [s, t]$ ,  $|\sum_{i \in I} (a_i - b_i)| = |S_t - S_{s-1}| < 1$ . Therefore, the supremum of the optimal rounding error  $Dist_\infty^\mathcal{I}(\mathbf{a}, \mathbf{b})$  is at most 1 for any family  $\mathcal{I}$  of intervals. On the other hand, there is an example that the supremum becomes 1 even if each interval has length 2.

**Proposition 1.** *If we consider the family  $\mathcal{I}$  of all intervals of length 2, there exists an input sequence  $\mathbf{a}$  for which there is no binary sequence  $\mathbf{b}$  attaining  $Dist_\infty^\mathcal{I}(\mathbf{a}, \mathbf{b}) < 1 - 1/(n-1)$ .*

The proof is given in Appendix 1.

### 2.2 Finding an Optimal Rounding — Known Results

The error diffusion algorithm computes an optimal rounding of  $\mathbf{a}$  with respect to  $Dist_\infty^\mathcal{I}$  if  $\mathcal{I} = \{[1, i] : i = 1, 2, \dots, n\}$ . However, it does not always find an optimal rounding for a general  $l_\infty$  distance. Moreover, we would like to deal with weighted  $l_\infty$  distances. If  $\mathcal{I}$  consists of all intervals of length  $k$  for a constant  $k$ , it is relatively easy to design a linear time algorithm with respect to  $n$  by using Viterbi's algorithm [21] based on dynamic programming (in precise,  $O(2^k n)$  time using  $O(2^k n)$  space) to compute an optimal rounding of  $\mathbf{a}$ . The space complexity

can be reduced to  $O(2^k \sqrt{n} + n)$  while keeping the time complexity [5] and it can be further reduced to  $O(2^k + n)$  if we spend  $O(2^k n \log n)$  time (a similar technique is found in [3]). However, it is nontrivial to design an efficient algorithm which is polynomial both in  $n$  and  $k$ .

### 2.3 Polynomial Time Algorithms

In this subsection, we give a polynomial time algorithm for computing an optimal rounding of a sequence  $\mathbf{a}$  with respect to the weighted  $l_\infty$  distance  $Dist_\infty^{\mathcal{I}, w}$  for a general set  $\mathcal{I}$  of intervals and any weight function  $w$  on  $\mathcal{I}$ . Without loss of generality, we can assume that all entries of  $\mathbf{a}$  are in the open interval  $(0, 1)$ , since we can ignore integral entries to solve the problem. Since we consider the weighted  $l_\infty$  distance, an optimal rounding of  $\mathbf{a}$  is a binary sequence  $\mathbf{b} = (b_1, b_2, \dots, b_n)$  which is the solution to the following integer programming problem, where  $z$  corresponds to the optimal distance between  $\mathbf{a}$  and  $\mathbf{b}$ :

$$\begin{aligned} & \text{minimize } z \\ & \text{subject to } -z \leq w(I) \sum_{j \in I} (a_j - b_j) \leq z \quad (\forall I \in \mathcal{I}), \\ & \quad b_j \in \{0, 1\} \quad (\forall j \in \{1, 2, \dots, n\}). \end{aligned} \tag{1}$$

Recall that the weight function is always positive, i.e.  $w(I) > 0$  for any  $I \in \mathcal{I}$ . Since the variables  $b_1, \dots, b_n$  are all 0-1 valued, we can replace the inequality constraints (1) by

$$\min\{|w(I)^{-1}z + \sum_{j \in I} a_j|, n\} \geq \sum_{j \in I} b_j \geq \max\{[-w(I)^{-1}z + \sum_{j \in I} a_j], 0\} \quad (\forall I \in \mathcal{I}).$$

We introduce the variables  $x_0, \dots, x_n$  satisfying  $x_i - x_0 = b_1 + \dots + b_i$  for  $i \in \{1, 2, \dots, n\}$ . For each interval  $I$ , the indices of its first entry and last entry are denoted by  $s(I)+1$  and  $t(I)$ ; In other words,  $I = [s(I)+1, t(I)] = (s(I), t(I))$ .

Then the above problem is transformed into the following problem

$$\begin{aligned} & \text{minimize } z \\ & \text{subject to } x_{t(I)} - x_{s(I)} \leq \min\{|w(I)^{-1}z + \sum_{j \in I} a_j|, n\} \quad (\forall I \in \mathcal{I}), \\ & \quad x_{s(I)} - x_{t(I)} \leq -\max\{[-w(I)^{-1}z + \sum_{j \in I} a_j], 0\} \quad (\forall I \in \mathcal{I}), \\ & \quad x_j - x_{j-1} \leq 1 \quad (\forall j \in \{1, 2, \dots, n\}), \\ & \quad x_{j-1} - x_j \leq 0 \quad (\forall j \in \{1, 2, \dots, n\}), \\ & \quad x_j \text{ is an integer} \quad (\forall j \in \{0, 1, 2, \dots, n\}). \end{aligned} \tag{2}$$

For the time being, we are concentrated on the decision problem: checking the existence of a vector  $(x_0, x_1, \dots, x_n)$  satisfying the above constraints when  $z$  is fixed. It is discussed later how to find the optimal value of  $z$ .

From the above integer programming formulation, The decision problem is an integer programming on a system of difference constraints, and it is well-known that the problem is transformed to that of detecting a negative cycle in a

graph [11]. In order to make the paper self-contained, we give the construction of the graph:

Let  $H = (N, E)$  be a directed graph with a vertex set  $N = \{0, 1, 2, \dots, n\}$  and an arc set  $E = E_1 \cup E_2 \cup E_3 \cup E_4$  where

$$\begin{aligned} E_1 &= \{(s(I), t(I)) : I \in \mathcal{I}\}, \\ E_2 &= \{(t(I), s(I)) : I \in \mathcal{I}\}, \\ E_3 &= \{(0, 1), (1, 2), \dots, (n-1, n)\}, \\ E_4 &= \{(1, 0), (2, 1), \dots, (n, n-1)\}. \end{aligned}$$

The arc weight  $w_{ij}$  of an arc  $(i, j) \in E$  is defined by

$$w_{i,j} = \begin{cases} \min\{\lfloor w(I)^{-1}z + \sum_{j \in I} a_j \rfloor, n\} & (\text{for } (i, j) = (s(I), t(I)) \in E_1), \\ -\max\{\lceil -w(I)^{-1}z + \sum_{j \in I} a_j \rceil, 0\} & (\text{if } (i, j) = (t(I), s(I)) \in E_2), \\ 1 & (\text{if } (i, j) \in E_3), \\ 0 & (\text{if } (i, j) \in E_4), \end{cases}$$

where the variable  $z$  is fixed. The arc sets  $E_1, E_2, E_3, E_4$  correspond to the constraints (2), (3), (4), (5), respectively. Note that the graph has  $O(n + |\mathcal{I}|)$  arcs.

A negative cycle is an elementary directed cycle  $C$  satisfying that the total sum of the arc weights in  $C$  is negative. The detection of a negative cycle in  $H$  can be done in  $O(n^{0.5}m \log(n\Gamma))$  time by using Gabow-Tarjan's scaling algorithm for assignment problem [4], where  $m$  is the number of edges and  $\Gamma$  is the maximum weight. By definition,  $m = O(n + |\mathcal{I}|)$  and  $\log(n\Gamma) = O(\log n)$  in our graph.

If there exists a negative cycle in  $H$ , then the inequality system (2), (3), (4), (5) is infeasible. On the other hand, if the graph contains no negative cycle, shortest path length  $x_i^*$  from the vertex  $i$  to  $n$  is well-defined and integer valued, and the vector  $(x_i^*)_{i=1,2,\dots,n}$  satisfies the inequalities (2), (3), (4), and (5). The radix heap implementation of Dijkstra's algorithm [1] finds the path lengths in  $O(m + n \log(n\Gamma)) = O(|\mathcal{I}| + n \log n)$  time.

Now, we discuss the method to find the optimal value of  $z$  (i.e., smallest  $z$  causing no negative cycle). We employ the ordinary binary search technique. Each edge weight is represented by a step function with respect to  $z$ . Thus, we only need to consider the break points of the step functions. If we define  $q(h, I) = w(I)(h + 0.5 + \sum_{j \in I} a_j)$  for an interval  $I$  and an integer  $h$ , the set  $Q = \{q(h, I) | I \in \mathcal{I}, -n \leq h \leq n\}$  contains all the break points. By applying binary search technique (with some care), we can find the optimal value of  $z$  by executing the above negative cycle detecting algorithm  $O(\log n|\mathcal{I}|) = O(\log n)$  times with additional  $O((n + |\mathcal{I}|) \log n)$  time for each search process. Thus we can find the optimal value of  $z$  in  $O(n^{0.5}(n + |\mathcal{I}|) \log^2 n)$  time in total. Hence, we have the following theorem:

**Theorem 1.** *An optimal rounding of a sequence with respect to the distance  $Dist_{\infty}^{\mathcal{I}, w}$  can be computed in  $O(n^{0.5}(n + |\mathcal{I}|) \log^2 n)$  time. The space requirement is  $O(n + |\mathcal{I}|)$ .*

We can design a better algorithm if each interval is short.

**Theorem 2.** *An optimal rounding of a sequence can be computed in  $O(k^2n \log n)$  time using  $O(n + k^2 + |\mathcal{I}|)$  space if the family  $\mathcal{I}$  is a set consisting of intervals of length at most  $k$ .*

*Proof.* First, we give an  $O(k^2n)$  time algorithm for checking the existence of negative cycles of  $H$ . For each index  $p \in \{0, 1, 2, \dots, n\}$ , the subgraph of  $H$  induced by the vertices  $\{0, 1, \dots, p\}$  is denoted by  $H_p$ . It is clear that  $H_p$  is strongly connected. For each triplet of vertices  $(i, j, p)$  satisfying  $i, j \in \{p - k, \dots, p\}$  and  $k \leq p$ ,  $d(i, j, p)$  denotes the shortest path length from  $i$  to  $j$  in the graph  $H_p$  when  $H_p$  does not contain any negative cycle. We first find a negative cycle in  $H_k$ , if it exists. If the graph  $H_k$  does not contain any negative cycle, we calculate the values  $\{d(i, j, k) \mid i, j \in \{0, \dots, k\}\}$  by using an all-pairs shortest path algorithm. Gabow-Tarjan's algorithm for assignment problems solves the negative cycle detection problem in  $O(k^{0.5}k^2 \log n) = O(k^2n)$  time. If we transform arc weights by using an optimal dual solution to the assignment problem obtained by Gabow-Tarjan's algorithm, we only need to solve an all-pairs shortest path problem defined on a network with non-negative arc weights. By applying the radix heap implementation of Dijkstra's algorithm  $k$  times, the computational requirement is bounded by  $O(k(k + |\mathcal{I}| + k \log n)) = O(k^2n)$  time, since  $|\mathcal{I}| = O(nk)$ .

Suppose that  $H_p$  has no negative cycle, and we have already computed  $\{d(i, j, m) \mid i, j \in \{0, \dots, m\}\}$  for all  $m \leq p$ . If the graph  $H_{p+1}$  has no negative cycle. Then we can calculate the values  $\{d(i, j, p+1) \mid i, j \in \{p - k + 1, \dots, p + 1\}\}$  from the values  $\{d(i, j, p) \mid i, j \in \{p - k, \dots, p\}\}$  easily. It is because, a shortest path  $P$  from  $i$  to  $j$  in  $H_{p+1}$  satisfies one of the following two conditions; (1)  $P$  is contained in  $H_p$ , or (2)  $P$  is partitioned into a sequence of three subpaths  $(P_1, P_2, P_3)$  such that  $P_1$  and  $P_3$  are contained in  $H_p$  and  $P_2$  is a path consisting of two arcs  $a_1$  and  $a_2$  where  $a_1$  and  $a_2$  shares the vertex  $p + 1$ . Since both the in-degree and out-degree of each vertex are bounded by  $O(k)$ , we can calculate the values  $\{d(i, j, p + 1) \mid i, j \in \{p - k + 1, \dots, p + 1\}\}$  using the values  $\{d(i, j, p) \mid i, j \in \{p - k, \dots, p\}\}$  in  $O(k^2)$  time if we maintain the values  $\{d(i, j, p) \mid i, j \in \{p - k, \dots, p\}\}$  by an  $(k + 1) \times (k + 1)$  array.

On the other hand, if  $H_{p+1}$  has a negative cycle  $C^*$ ,  $C^*$  must contain the vertex  $p + 1$ . Since we have the shortest path length in  $H_p$  between each (ordered) pair of vertices in  $\{p - k, p - k + 1, \dots, p\}$  and each vertex adjacent to  $p + 1$  in  $H_{p+1}$  must be in  $\{p - k, p - k + 1, \dots, p\}$ , we can check the existence of a negative cycle easily. Since the degree of each vertex is bounded by a constant, we can check the existence of a negative cycle in constant time. The above observations imply that we can check the existence of a negative cycle of  $H$  in  $O(k^2n)$  time.

When the variable  $z$  is fixed and the graph  $H$  does not have any negative cycle, we can find the shortest path lengths  $x_i^*$  for  $i = 1, 2, \dots, n - 1$  in  $O(k^2n)$  time by using a similar argument. Replacing the  $O(n^{0.5}(n + |\mathcal{I}|) \log n)$  time algorithm for negative cycle detection with the above algorithm leads to an  $O(k^2n \log n)$  time algorithm.

### 3 Matrix-Rounding Problem

#### 3.1 Discrepancy Problem

The following theorem is well known [IO]:

**Theorem 3.** *The inhomogeneous discrepancy of a  $[0, 1]$ -valued  $n \times n$  matrix with respect to the family of all rectangular regions is  $O(\log^3 n)$  and  $\Omega(\log n)$ . The same bounds hold for the inhomogeneous discrepancy for the family of all rectangular regions containing the left-upper corner entry of the matrix.*

We are interested in the matrix-rounding with respect to the set  $\mathcal{W}_k$  of all  $k \times k$  square regions. The following proposition is obtained in a straightforward manner from the theorem above:

**Proposition 2.** *The inhomogeneous discrepancy with respect to  $\mathcal{W}_k$  is  $O(\log^3 k)$  and  $\Omega(\log k)$ . Indeed, these bounds also hold for the union  $\cup_{j=1}^k \mathcal{W}_j$ .*

*Proof.* Without loss of generality, we assume that  $k$  divides  $n$ , and subdivide the grid into  $(n/k)^2$  subgrids of size  $k \times k$ . Then, each element of  $\cup_{j=1}^k \mathcal{W}_j$  is a union of four rectangles in subgrids, and hence we have an  $O(\log^3 k)$  bound from Theorem 3.

In order to show the lower bound, consider a  $2k \times 2k$  matrix  $A$  whose lower-right quarter is an  $\Omega(\log k)$  instance for the discrepancy with respect to all rectangular regions in the quarter containing the entry  $a_{k+1,k+1}$ . The remaining part of  $A$  is filled with zero entries. For any given rectangular region  $R$  containing  $a_{k+1,k+1}$  in the lower-right quarter, there exists a region in  $\mathcal{W}_k$  consisting of  $R$  and zero entries. Hence we have the lower bound for  $\mathcal{W}_k$ .

We remark that a polynomial time algorithm for computing a rounding with an  $O(\log^4 k)$  discrepancy can be designed based on the proof of Theorem 6.13 in [IO]. This is theoretically better than the popular two-dimensional error diffusion algorithm, for which the rounding error can become  $k$  (see Appendix 2).

For the family  $\mathcal{W}_2$  consisting of all  $2 \times 2$  square regions, there exists an instance  $A$  that the discrepancy is exactly 1. However, the authors do not know whether there exists an instance requiring  $\text{Dist}_{\infty}^{\mathcal{W}_2}(A, B) > 1$  or not. It is easy to show that the inhomogeneous discrepancy with respect to  $\mathcal{W}_2$  is at most 2; indeed, the checkerboard binary matrix  $C$  satisfies  $\text{Dist}_{\infty}^{\mathcal{W}_2}(A, B) \leq 2$  for any input matrix  $A$  simultaneously. However, it is nontrivial to give a better upper bound; We can prove the following result (the proof is involved, and omitted in this version):

**Theorem 4.** *For any  $[0, 1]$  valued matrix  $A$ , there exists a binary matrix  $B$  satisfying that  $\text{Dist}_{\infty}^{\mathcal{W}_2}(A, B) \leq 1.75$ .*

### 3.2 NP-hardness of Computing an Optimal Matrix Rounding

In this subsection, we prove the following theorem:

**Theorem 5.** *For any  $\epsilon > 0$ , it is NP-hard to decide whether the optimal rounding error of a given matrix  $A$  is greater than  $1 - \epsilon$  or less than  $1/2 + \epsilon$  with respect to the distance  $Dist_{\infty}^{\mathcal{W}_2}$ .*

For simplicity, we write  $Dist(A, B)$  for the distance  $Dist_{\infty}^{\mathcal{W}_2}(A, B)$ . Each entry of our hardness instance  $A$  has one of three values: 0, 1, and  $1/2$ , where we use a convention that we can round 0 to 1 and 1 to 0. To make the proof mathematically formal, we should replace 0 and 1 with  $\delta$  and  $1-\delta$  for an infinitesimally small positive number  $\delta$  satisfying  $\delta < \epsilon/4$ . This is the reason why  $\epsilon$  appears in the statement of the theorem. By using the above mentioned convention, we ignore  $\epsilon$  and  $\delta$  in the proof.

We prepare some useful definitions and a lemma. A zero-entry of  $A$  is called an *absolute-zero* entry if it is contained in a  $2 \times 2$  square such that all of its entries are zeros. A pair of two  $1/2$  entries is called a *good pair* if there exists a  $2 \times 2$  square region consisting of the pair and two absolute-zero entries. The following lemma is immediate:

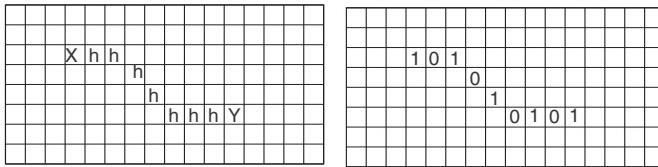
**Lemma 1.** *If  $Dist(A, B) \leq 1/2$ , each absolute-zero entry must become 0 in  $B$ . Moreover, each good pair must become a pair of 0 and 1 in  $B$ .*

We prove the theorem by using a reduction from the planar 3-SAT problem [15]. An instance of planar 3-SAT is a Boolean expression  $E = E_1 \wedge E_2 \wedge \dots \wedge E_m$  where each clause  $E_j$  contains at most three literals which are variables or their negations and a planar graph is defined by the vertex set  $\{E_1, E_2, \dots, E_m, u_1, u_2, \dots, u_q\}$  and the edge set  $\{(E_i, u_j) | E_j \text{ contains } u_j \text{ or } \bar{u}_j\}$ . The nodes  $E_i$  ( $i = 1, 2, \dots, m$ ) are called the *clause nodes*, while the nodes  $u_j$  ( $j = 1, 2, \dots, q$ ) are called the *literal nodes*. Then, the problem is to decide whether there exists an assignment  $F \subseteq \{u_1, \bar{u}_1, u_2, \bar{u}_2, \dots, u_q, \bar{u}_q\}$  making the expression  $E$  true.

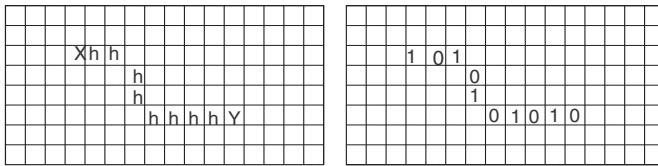
A polynomial time reduction from a planar 3-SAT problem to the corresponding optimal halftoning is established as follows: Suppose that we are given a Boolean expression  $E$  of the above form together with a planar graph defined above. It is well-known that the planar graph representing the expression  $E$  can be replaced with a graph  $G$  embedded in a pixel grid of size polynomial in the total number of clauses and variables. Two pixels  $(i, j)$  and  $(i', j')$  in the grid is called adjacent if  $|i - i'| \leq 1$  and  $|j - j'| \leq 1$ . In other words, they are located in a common region in  $\mathcal{W}_2$ . Each edge of the graph  $G$  is represented by a series of adjacent pixels. There are three kinds of nodes of  $G$ , literal nodes, branching nodes and clause nodes. We can assume that there is enough grid space between each pair of nodes. In the following, we further modify the graph  $G$  such that the SAT assignment information is represented by using a  $[0, 1]$ -valued matrix  $A$ , such that  $E$  is satisfiable if the optimal rounding error of  $A$  is at most  $1/2$ , otherwise it is at least 1.

For each variable node  $(i, j)$  associated with  $u_s$ , we set  $a_{i,j} = 1/2$ . If we assign 0 to the variable  $u_s$ , then  $b_{i,j} = 0$ , else  $b_{i,j} = 1$ . An edge between two nodes is a path consisting of adjacent half-entries. Moreover, each pair of adjacent pixels must form a good pair. Figure 1 shows our gadget representing an edge of  $G$  between two nodes  $X$  and  $Y$ . For convenience' sake, we omit to write zero-entries in the figures, and also each half-entry is represented by an  $h$  (meaning "half"). The nodes  $X$  and  $Y$  also have values  $1/2$ . Note that the direction of the edge can be bent to any of four possible slopes if we have a sufficient open space.

From Lemma 1, if there exists an approximation  $B$  of  $A$  with distance (at most)  $1/2$ , there are only two possible assignments. One is shown in Figure 1, and the other is its opposite. This means that the value of  $b_Y$  of  $B$  at  $Y$  is uniquely determined by  $b_X$ . In the case of Figure 1  $b_Y = b_X$ . However, we can define another path shown in Figure 2 forcing  $b_Y = \bar{b}_X$ ; thus, it creates the negation of a variable. Indeed, we can make both an odd-length path and an even-length path between two nodes to control the assignment of the literal in each clause. Our gadget representing a branching point node of  $G$  is illustrated

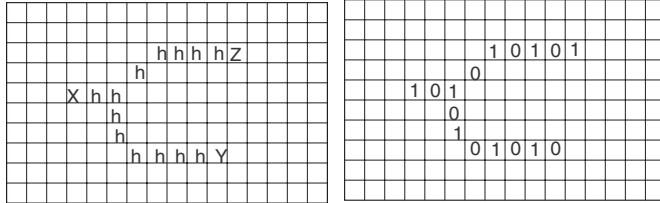


**Fig. 1.** A gadget representing an edge of  $G$  which makes  $b_X = b_Y$ .



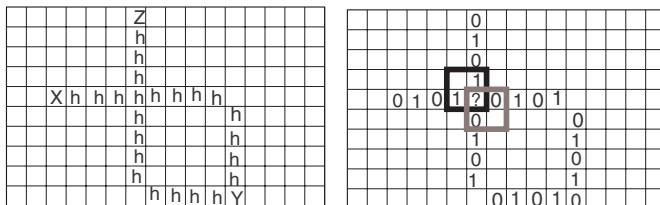
**Fig. 2.** A gadget representing an edge of  $G$  which makes  $b_Y$  to be the negation of  $b_X$ .

in Figure 3. Note that all zero-entries of the input matrix (left-side drawing) are absolute-zero entries. The rest to show is that we can simulate a clause node in the planar 3-SAT instance. Let the clause be  $x \vee y \vee z$ , where  $x, y$  and  $z$  are literals or their negations. First, we make a weaker gadget, which corresponds to  $(x \vee y \vee z) \wedge (\bar{x} \vee \bar{y} \vee \bar{z})$  (not-all-equal-3SAT clause). The left drawing of Figure 4

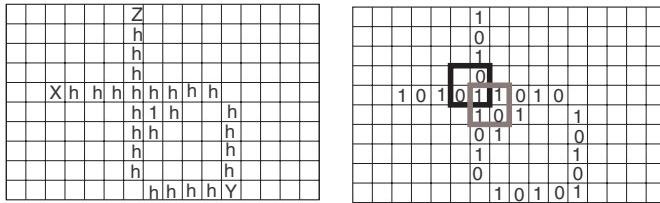
**Fig. 3.** A gadget for a branching node.

illustrates the assignment of  $1/2$  entries in the input matrix  $A$ . The values of the matrix  $B$  at  $X$ ,  $Y$ , and  $Z$  correspond to the Boolean values of  $x$ ,  $y$ , and  $z$ , respectively.

If  $dist(A, B) < 1$ , once we fix the values at  $X$ ,  $Y$ , and  $Z$  in  $B$ , all values except the  $h$  at the center crossing are uniquely determined. If  $X = Y = Z = 0$ , the matrix  $B$  is the one illustrated in the right drawing of Figure 4. We will show that there is no possible assignment at the pixel  $p$  with the ? mark if  $dist(A, B) < 1$ : Since the  $2 \times 2$  matrix containing  $p$  as its south-east corner has the entry sum  $3/2$  in  $A$ , its entry sum in  $B$  must be either 1 or 2. Hence, the value at  $p$  must be 0 in  $B$ . However, the  $2 \times 2$  matrix contains  $p$  as its north-west corner also has the entry sum  $3/2$  in  $A$ , and hence the value at  $p$  must be 1 in  $B$ . This is a contradiction. We can see that  $X = Y = Z = 1$  is another impossible assignment to make  $dist(A, B) < 1$ ; on the other hand, for all other assignments of  $X$ ,  $Y$ , and  $Z$ , we can find a rounding  $B$  satisfying  $dist(A, B) = 1/2$ . Next, we modify the above gadget to the matrix in the left

**Fig. 4.** A gadget for a not-all-equal-3SAT clause node.

drawing of Figure 5. It is easy to see that if  $X = Y = Z = 0$ , there is no  $B$  such that  $Dist(A, B) < 1$ . However, if  $X = Y = Z = 1$ , the right-hand side drawing shows that it is possible to make  $Dist(A, B) = 1/2$ . A key difference is that we have a one-entry in  $A$ , which is permitted to become 0 in  $B$  without violating the distance condition. We have constructed all required gadgets, and



**Fig. 5.** A gadget for a clause node representing  $X \vee Y \vee Z$ .

thus proved that the planar 3SAT instance  $E$  is satisfiable if and only if there exists a Boolean matrix  $B$  satisfying  $\text{dist}(A, B) \leq 1/2$ . On the other hand, if  $E$  is not satisfiable, then  $\text{Dist}(A, B) \geq 1$ . Thus, we have proved the theorem.

**Corollary 1.** *For any  $k \geq 1$ , it is NP-hard to decide whether an optimal rounding  $B$  for an input  $A$  satisfies  $\text{Dist}_{\infty}^{\mathcal{W}_{2k}}(A, B) \geq 1 - \epsilon$  or  $\text{Dist}_{\infty}^{\mathcal{W}_{2k}}(A, B) \leq 1/2 + \epsilon$ .*

*Proof.* Let  $A = (a_{i,j})$  be the instance of 2-approximate hardness for  $\mathcal{W}_2$  constructed in the proof of Theorem 5. Then, we define a  $kn \times kn$  matrix  $C = (c_{i,j})$  as follows: We call an entry  $c_{i,j}$  *special* if both of  $i$  and  $j$  are divisible by  $k$ . The values of special entries are defined by  $c_{sk,tk} = a_{s,t}$  ( $1 \leq s \leq n, 1 \leq t \leq n$ ). Other entries are defined to be zero entries. For any  $W \in \mathcal{W}_{2k}$ ,  $W$  contains exactly four special entries, which correspond to entries of  $A$  located in a region in  $\mathcal{W}_2$ . It can be seen that flipping a non-special entry to 1 forces the rounding error to be at least  $1 - \epsilon$ . Therefore, we have the theorem.

## 4 Concluding Remarks

We have considered sequence-rounding and matrix-rounding problems. The sequence-rounding problem has been solved well; in particular, the optimization problem can be solved in polynomial time for the weighted  $l_{\infty}$  distance. As for the matrix-rounding problem, many problems are left open. We especially want to design a nice approximation algorithm for the matrix-rounding problem with respect to  $\text{Dist}_{\infty}^{\mathcal{W}_k}$ .

## References

1. R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan: “Faster algorithms for the shortest path problems,” *Journal of ACM*, 37 (1990), pp. 213–223.
2. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows, Theory Algorithms and Applications*, Princeton Hall, 1993.
3. T. Asano, D. Z. Chen, N. Katoh, and T. Tokuyama: “Polynomial-time solutions to image segmentation,” *Proc. of the 7th ACM-SIAM Symposium on Discrete Algorithms* (1996), pp. 104–113.

4. T. Asano, N. Katoh, H. Tamaki, and T. Tokuyama: "Convertibility among grid filling curves," *Proc. ISAAC98, Springer LNCS 1533* (1998), pp. 307–316.
5. T. Asano, D. Ranjan and T. Roos: "Digital halftoning algorithms based on optimization criteria and their experimental evaluation," *IEICE Trans. Fundamentals*, Vol. E79-A (1996), No. 4, pp. 524–532.
6. B. E. Bayer: "An optimum method for two-level rendition of continuous-tone pictures," *Conference Record, IEEE International Conference on Communications*, 1 (1973), pp. (26-11)–(26-15).
7. Z. Baranyai, "On the factorization of the complete uniform hypergraphs", in *Infinite and Finite Sets*, eds. A. Hanaj, R. Rado and V. T. Sós, *Colloq. Math. Soc. János Bolyai* **10** (1974), pp.91-108.
8. B. Bollobás, *Combinatorics*, Cambridge University Press, 1986.
9. J. Beck and W. Chen, *Irregularities of Distribution*, Cambridge University Press, Cambridge, 1987.
10. J. Beck and V. T. Sós, *Discrepancy Theory*, in *Handbook of Combinatorics* Volume II, (ed. R. Graham, M. Grötschel, and L. Lovász) 1995, Elsevier.
11. T. H. Cormen, C. E. Leiserson, R. L. Rivest: *Introduction to Algorithms*, MIT Press, 1989.
12. Y. Crama, P. Hansen and B. Jaumard: "The basic algorithm for pseudo-Boolean programming revisited," *Discrete Applied Mathematics*, 29 (1990), pp. 171–185.
13. R. W. Floyd and L. Steinberg: "An adaptive algorithm for spatial gray scale," *SID 75 Digest, Society for Information Display* (1975), pp. 36–37.
14. H. N. Gabow and R. E. Tarjan: "Faster scaling algorithms for network problems," *SIAM J. Comp.*, 18 (1989), pp. 1013–1036.
15. M. R. Garey and D. S. Johnson: *Computers and Intractability: A guide to theory of NP hardness*, Freeman and Company, 1979.
16. D. E. Knuth: "Digital halftones by dot diffusion," *ACM Trans. Graphics*, 6-4 (1987), pp. 245–273.
17. J. O. Limb: "Design of dither waveforms for quantized visual signals," *Bell Syst. Tech. J.*, 48-7 (1969), pp. 2555–2582.
18. B. Lippel and M. Kurland: "The effect of dither on luminance quantization of pictures," *IEEE Trans. Commun. Tech.*, COM-19 (1971), pp.879-888.
19. V. Rödl and P. Winkler: "Concerning a matrix approximation problem", *Crux Mathematicorum*, 1990, pp. 76–79.
20. R. Ulichney: *Digital halftoning*, MIT Press, 1987.
21. A. J. Viterbi: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm *IEEE Transactions on Information Theory* Vol. IT-13 (1967), pp.260–267.

## Appendix 1. Proof of Proposition $\square$

*Proof.* Consider a sequence  $\mathbf{a} = (a_i)$ ,  $i = 1, 2, \dots, 2p$ , defined by  $0, 1/(4p-1), (4p-3)/(4p-1), 3/(4p-1), (4p-5)/(4p-1), \dots, (2p-2)/(4p-1), 2p/(4p-1)$ , which satisfies  $a_{2j} + a_{2j+1} = (4p-2)/(4p-1)$  and  $a_{2j+1} + a_{2j+2} = 4p/(4p-1)$  for  $1 \leq j \leq p-1$ . From the construction it is obvious that there is a unique  $\mathbf{b}$  approximating  $\mathbf{a}$  with distance less than  $(4p-2)/(4p-1)$ . Since we consider the intervals of length 2, this means that  $|a_i + a_{i+1} - b_i - b_{i+1}| < (4p-2)/(4p-1)$  for  $i = 1, 2, \dots, 2p-1$ . Indeed,  $\mathbf{b} = 0, 0, 1, 0, 1, 0, 1, 0, \dots, 1, 0, 1$ , which is an alternating sequence except the first two zeros.

Let  $\mathbf{a}'$  be the reversed sequence of  $\mathbf{a}$ , and consider the concatenation of  $\mathbf{a}$  and  $\mathbf{a}'$ . Naturally, the only possible approximation must be the concatenation of  $\mathbf{b}$  and its reverse  $\mathbf{b}'$ . However, the sequences meet at the middle of the whole input sequence so that  $2p/(4p-1)$  and  $2p/(4p-1)$  are adjacent, and the corresponding outputs are 1 and 1. Thus, the difference of the entry sums in this neighborhood is  $2 - 4p/(4p-1) = (4p-2)/(4p-1)$ . Hence, it is impossible to approximate it within the distance of  $(4p-2)/(4p-1)$ , which proves the proposition.

## Appendix 2. Performance of an Error Diffusion Algorithm.

A popular practical method in digital halftoning is the error diffusion algorithm (we have already seen its one-dimensional version): Scan the matrix array  $A$  in the scan-line order (i.e., scan row-wise from top row to bottom row, from left to right in each row), and greedily round the entries of  $A$  into binary values propagating the remaining error at each visited entry to its unvisited neighbor entries. The outline of the algorithm is as follows: We use four parameters  $\alpha, \beta, \gamma$ , and  $\delta$  satisfying  $\alpha + \beta + \gamma + \delta = 1$ . At first,  $\text{error}(i, j)$  is initialized as 0 for each pixel  $(i, j)$  in the grid. When we visit the pixel  $(i, j)$ ,  $b_{i,j}$  is obtained by rounding  $a_{i,j} + \text{error}(i, j)$  to the nearer binary value. Now, compute  $\text{rem}(i, j) = a_{i,j} + \text{error}(i, j) - b_{i,j}$ , which is the error remaining at  $(i, j)$ , and distribute the error values to its unvisited neighbors with predetermined weights. Formally, those errors are updated as follows:  $\text{error}(i, j+1) := \text{error}(i, j+1) + \alpha \text{rem}(i, j)$ ,  $\text{error}(i+1, j) := \text{error}(i+1, j) + \beta \text{rem}(i, j)$ ,  $\text{error}(i+1, j+1) := \text{error}(i+1, j+1) + \gamma \text{rem}(i, j)$ , and  $\text{error}(i+1, j-1) := \text{error}(i+1, j-1) + \delta \text{rem}(i, j)$ . We need some care for pixels near the boundary of  $G$ , but we ignore it here for simplicity.

**Proposition 3.** Suppose that  $B$  is the output of the error diffusion algorithm for an input  $A$ . Then, for the region family  $\mathcal{W}_k$ ,  $\text{Dist}_{\infty}^{\mathcal{W}_k}(A, B) \leq k + (k-1)(\gamma + \delta)$ . Moreover, there is an instance  $A$  such that the above distance exceeds  $k + (k-1)(\gamma + \delta) - \epsilon$  for any  $\epsilon > 0$  if we apply the error diffusion algorithm.

*Proof.* It is observed that  $-0.5 \leq \text{rem}(i, j) < 0.5$  holds for  $1 \leq i \leq n$  and  $i \leq j \leq n$ . Fix a  $k \times k$  square  $R$  in  $G$ . Let  $S_{in}$  be a set of entries outside  $R$  from which error is directly propagated to some entry in  $R$ . Also, let  $S_{out}$  be a set of entries in  $R$  from which error is directly propagated to some entry outside  $R$ . Consider the total error propagated into  $R$  and also the total error propagated out of  $R$ , we have  $|A(R) - B(R)| \leq k + (k-1)(\gamma + \delta)$ . On the other hand, we can manage to create the input attaining  $\text{rem}(i, j) = -1/2$  for each  $(i, j) \in S_{in}$ , and  $\text{rem}(i', j') > 1/2 - \epsilon/k$  for each  $(i', j') \in S_{out}$ . This gives the lower bound.

# On the Complexity of the Sub-permutation Problem

Shlomo Ahal<sup>1</sup> and Yuri Rabinovich<sup>2</sup>

<sup>1</sup> Math. Department, Ben-Gurion University of the Negev, Beer Sheva, Israel.

<sup>2</sup> CS Dept., Haifa University, Haifa, Israel, [yuri@cs.haifa.ac.il](mailto:yuri@cs.haifa.ac.il)

**Abstract.** We study various computational aspects of the problem of determining whether a given order contains a given sub-order. Formally, given a permutation  $\pi$  on  $k$  elements, and a permutation  $\sigma$  on  $n > k$  elements, the goal is to determine whether there exists a strictly increasing function  $f$  from  $[1..k]$  to  $[1..n]$  which is order preserving, i.e.,  $f$  satisfies  $\sigma(f(i)) > \sigma(f(j))$  whenever  $\pi(i) > \pi(j)$ . We call this decision problem the *Sub-Permutation Problem*.

The study falls into two parts. In the first part we develop and analyze an algorithm (or, rather, an algorithmic paradigm) for this problem. We show that the complexity of this algorithm is at most  $O(n^{1+C(\pi)})$ , where  $C(\pi)$  is a naturally defined function of the permutation  $\pi$ .

In the second part we study  $C(\pi)$ . In particular, we show that  $C(\pi) \leq 0.35k + o(k)$ , implying that the complexity of the Sub-Permutation problem is  $O(c_k + n^{0.35k+o(k)})$ . On the other hand, we prove that for most  $\pi$ 's,  $C(\pi) = \Omega(k)$ , establishing a lower bound for our algorithm. In addition, we develop a fast polylogarithmic approximation algorithm for computing  $C(\pi)$ , and bound the value of this parameter for some interesting families of permutations.

## 1 Introduction

The question studied in this paper belongs to the vast family of questions dealing with finding a specified substructure within a given (large) structure. Our structures here are permutations, or, rather, order types of finite numerical sequences.

**Definition 1.** Let  $\pi$  be a permutation on  $k$  elements, and  $\sigma$  a permutation on  $n$  elements, where  $n \geq k$ . A function  $f : [1..k] \mapsto [1..n]$  will be called an embedding of  $\pi$  into  $\sigma$  if it is

1. Strictly increasing, i.e.,  $f(i) > f(j)$  whenever  $i > j$ ;
2. Order preserving, i.e.,  $\sigma(f(i)) > \sigma(f(j))$  whenever  $\pi(i) > \pi(j)$ .

In other words,  $\pi$  is embeddable in  $\sigma$  if it has the same order type as a subsequence of  $\sigma$  obtained by erasing some of  $\sigma$ 's entries. In what follows, we shall use the notions “ $\pi$  is embeddable in  $\sigma$ ” and “ $\pi$  is a sub-permutation of  $\sigma$ ” interchangeably, and denote this by  $\pi \hookleftarrow \sigma$ .

Families of permutations with a fixed set of forbidden sub-permutations occur naturally e.g., in the study of permutations obtainable by using a single stack, or, more notably, in the well known open conjecture of Stanley-Wilf, claiming that for a fixed  $\pi$  the size of the families  $\mathcal{F}_n = \{\sigma \in S_n \mid \pi \not\hookrightarrow \sigma\}$ , grows at most exponentially in  $n$ . Despite a considerable research effort, (see, e.g., [3],[1]), the conjecture remains largely open. It seems that one of the main obstacles towards proving it is the lack of convenient criterions for establishing whether  $\sigma \in \mathcal{F}_n$  or not. In particular, the problem of checking whether  $\pi \hookrightarrow \sigma$  holds, is computationally difficult: it was shown to be NP-complete in [4].

In this paper we concentrate on the following version of the latter question, to be called the *Sub-Permutation problem*, (*SP*):

*Let  $\pi \in S_k$  be fixed. Given an input  $\sigma \in S_n$ , determine whether the relation  $\pi \hookrightarrow \sigma$  holds.*

To distinguish between  $\pi$  and  $\sigma$ , we shall call the former the *structure* permutation, and the later the *goal* permutation.

The above problem is, of course, polynomially tractable, and the brute force approach (checking all the sub-permutations of  $\sigma$  of size  $k$ ) gives an  $O(k \cdot n^k / k!)$  upper bound. A closer look reveals that the difficulty of the *SP* problem crucially depends on the structure permutation  $\pi$ , and while, for instance,  $\pi = (12\dots k)$  leads to a problem of complexity  $O(kn)$ , there is no obvious way to substantially improve upon the brute-force upper bound for a random  $\pi \in S_k$ .

What is the correct complexity of a *SP* problem for a given  $\pi$ ? To our best knowledge, this question was not addressed so far in the literature. We develop an algorithm, whose performance is bounded by  $O(c_k + kn^{1+C(\pi)})$ , where  $C(\pi)$  is a naturally defined function of  $\pi$ . We prove a general  $0.35k + o(k)$  upper bound on  $C(\pi)$ , but also, unfortunately, show that for most  $\pi \in S_k$  it holds  $C(\pi) = \Omega(k)$ . This, however, is not always the case. We present a number of a naturally defined classes of permutations  $\pi$  for which  $C(\pi)$  is  $O(\sqrt{k})$  and less.

While the exact computation of  $C(\pi)$  appears to be computationally difficult (and we do not know how to do it faster than in  $O(k2^k)$ ), we indicate how an  $O(\log^2 k)$  approximation of  $C(\pi)$  can be obtained in time polynomial in  $k$ .

Thus, we make first steps in the study of the fascinating question of the complexity of a *SP* problem as a function of the structure permutation  $\pi \in S_k$ . The most interesting related open question remains: can the *SP* problem can always be solved in time  $c_k n^{o(k)}$  for  $\pi \in S_k$ ?

## 2 The Generic Algorithm

We shall be mainly interested here in either finding a (single) embedding  $\pi \hookrightarrow \sigma$ , or concluding that no such embedding exists. However, after describing the procedure, we shall indicate how it can be modified in order to find *all* such embeddings.

Our approach is basically that of dynamic programming. The structure permutation  $\pi$  will be gradually *exposed*, and at each stage the corresponding tables will be suitably refined.

Consider (for the present, fixed) chain of subsets of  $[1..k]$ ,  $\emptyset = A_0 \subset A_1 \subset \dots \subset A_k = [1..k]$ , such that each  $A_i \setminus A_{i-1}$  consists of a single element. Setting  $\tau(i) = A_i \setminus A_{i-1}$ , we get a 1-1 function  $\tau : [1..k] \mapsto [1..k]$ , which we shall call the *order of exposure* corresponding to (and defining) the chain of subsets. The restrictions  $\pi_i$ 's of  $\pi$  to  $A_i$ 's form a chain of sub-permutations  $\emptyset = \pi_0 \hookrightarrow \pi_1 \hookrightarrow \dots \hookrightarrow \pi_k = \pi$ . Rigorously speaking, each  $\pi_i$  is a 1-1 function from  $A_i$  to  $[1..k]$ ; however, since we shall be interesting only in the order type of  $\pi_i$ , without a risk of confusion, we shall think of it as of permutation  $\pi_i \in S_i$ .

Roughly, our strategy is to create tables  $T_0, T_1, \dots, T_k$ , where  $T_i$  will store embeddings  $f_i : A_i \mapsto [1..n]$  of  $\pi_i$  into  $\sigma$ . In order to get  $T_i$ , we take  $T_{i-1}$  and extend, if possible, each embedding  $f_{i-1} \in T_{i-1}$  to a set of embeddings  $f_i$ .

How does one obtain legal extensions  $f_i$  of  $f_{i-1}$ ? Since  $f_i$  must agree with  $f_{i-1}$  on  $A_{i-1}$ , the question is where the new element  $\tau(i)$  can be mapped. The restrictions on the value of  $f_i(\tau(i))$  are:

**Monotonicity:** Let  $p_i^- \in A_{i-1}$  and  $p_i^+ \in A_{i-1}$  be, respectively, the next to the left and the next to the to right elements to  $\tau(i)$  in  $A_i$  (we view  $A_i$  as an ordered set). Then, in order to maintain monotonicity, it should hold

$$f_{i-1}(p_i^-) < f_i(\tau(i)) < f_{i-1}(p_i^+).$$

**Order Preservation:** Let  $q_i^- \in A_{i-1}$  and  $q_i^+ \in A_{i-1}$  be, respectively, the elements satisfying  $\pi_i(\tau(i)) - \pi_i(q_i^-) = 1$  and  $\pi_i(q_i^+) - \pi_i(\tau(i)) = 1$ . Then, to maintain the order preservation property, it should hold

$$\sigma(f_{i-1}(q_i^-)) < \sigma(f_i(\tau(i))) < \sigma(f_{i-1}(q_i^+)).$$

It is not hard to get convinced that the above restrictions on the value of  $f_i(\tau(i))$  are necessary and sufficient.

**Definition 2.** In what follows, we call  $\{p_i^-, p_i^+, q_i^-, q_i^+\} \subseteq A_{i-1}$  the significant elements for the  $i$ -th stage. The set of all possible values  $\{f_i(\tau(i))\}$  in a legal extension of a given  $f_{i-1}$  is completely determined by the values of  $f_{i-1}$  on the set of the significant elements for the  $i$ -th stage. Note that this set may contain less than four elements: e.g., for  $i = 1$ , it is empty.

Getting back to our strategy, it immediately becomes clear that saving in  $T_i$  the entire set of possible embeddings  $f_i : A_i \mapsto [1..n]$  is infeasible: the size of  $T_i$  can get as large as  $\binom{n}{4}$ , and we gain nothing compared to the brute force algorithm. How, then, could the content of  $T_i$  be condensed without incurring an information loss?

Let us first examine the situation in a particularly clear and simple case when  $\pi = (12..k)$ , and the exposure order  $\tau$  is  $1, 2, 3, \dots, k$  (or, equivalently,  $A_i = \{1, 2, \dots, i\}$  for  $i = 1, \dots, k$ ). Clearly, the only information about any particular  $f_i$  which will be used in the subsequent extensions is the value of  $f_i(i)$ . Thus, storing in  $T_i$  the values of  $f_i(i)$  alone, we get tables of size just  $O(n)$  which hold all the necessary information. The actual embedding can be reconstructed, if desired, using back-pointers where each  $f_i(i)$  points to one of its fathers  $f_{i-1}(i-1)$ .

For a better understanding of the nature of the gain in the above example, let us define for each  $i = 0, 1, \dots, k-1$  the set  $U_i \subseteq [1..k]$  of *unforgettable* elements:

**Definition 3.** Let  $U_i$ , the  $i$ -th set of unforgettable elements, be defined as the union of all the significant elements for stages  $i+1, \dots, k$ . That is,  
 $U_i = \cup_{j=i+1}^k \{p_j^-, p_j^+, q_j^-, q_j^+\}$ .

The definition of the significant elements immediately implies the following proposition:

**Proposition 1.** For any embedding  $f_i : A_i \mapsto [1..k]$ , the only elements of  $A_i$  whose value will ever be significant for the subsequent extensions of  $f_i$ , are  $U_i \cap A_i$ .

Therefore, following the usual logic of dynamic programming, it suffices to store in  $T_i$  only the values of  $f(i)$ 's on  $U_i \cap A_i$ . In fact, this is precisely what was done in the above simple example, since in that case  $U_i \cap A_i = \{i\}$ .

Before going on with the formal description of the emerging algorithm, let us have a different, clearer look at the system of sets  $\{U_i \cap A_i\}$ .

**Definition 4.** Let  $\pi \in S_k$  be a structure permutation. The incidence graph  $G_\pi$  is a undirected multi-graph on  $k$  vertices with two types of edges: blue ones and red ones. Formally,  $V(G_\pi) = [1..k]$  and  $E(G_\pi) = E_{blue}(G_\pi) \cup E_{red}(G_\pi)$ , where

$$E_{blue}(G_\pi) = \{ (i, j) \mid |i - j| = 1 \}; \quad E_{red}(G_\pi) = \{ (i, j) \mid |\pi(i) - \pi(j)| = 1 \} .$$

The following proposition establishes a surprising connection between  $U_i \cap A_i$  and the boundary of  $A_i$  in  $G_\pi$ :

**Proposition 2.** Define  $\partial_\pi A$ , the boundary of a subset  $A$  of  $V(G_\pi)$ , as  $\partial_\pi A = \{v \in A \mid \Gamma(v) \not\subseteq A\}$ , where  $\Gamma(v)$  is the set of neighbours of  $v$  in  $G_\pi$ . Then

$$U_i \cap A_i = \partial_\pi A_i .$$

*Proof.* Let  $v \in \partial_\pi A_i$ ; this means that  $v$  possesses a neighbour  $u \notin A_i$ . Let  $j > i$  be the stage when  $u$  is first exposed. It is readily checked that  $v$  is a significant element for the  $j$ -th step, and therefore  $v \in U_i$ .

Conversely, let  $v \in U_i \cap A_i$ . Assume  $v$  is significant for the  $j$ -th step,  $j > i$ . Then, clearly,  $v \in \partial_\pi A_j$ , and since  $A_i \subseteq A_j$ , this implies  $v \in \partial_\pi A_i$ . ■

We may now present an algorithm for the SP problem. The main data structure will be a sequence of tables  $\{T_i\}_{i=0}^k$ . The rows of each  $T_i$  will contain a numerical field per each element of  $\partial_\pi(A_i)$ , and an additional single field for a pointer. Each row in  $T_i$  will correspond to some embedding  $f_i : A_i \mapsto [1..n]$  of  $\pi_i$  into  $\sigma$ . The numerical fields will carry the values of  $f_i$  on  $\partial_\pi(A_i) = U_i \cap A_i$ , while the pointer will point to some  $f_{i-1}$  (a row of  $T_{i-1}$ ) whose extension resulted in  $f_i$ . The rows of  $T_i$  must all be different with respect to the numerical fields. Observe that  $T_k$  has a single row containing only the pointer, while while  $T_0$  is in fact an empty table.

**ALG-SP <sub>$\pi, \tau$</sub> ( $\sigma$ ):**

INPUT:  $\sigma \in S_n$  - a goal permutation ;  
 $\pi \in S_k$  - a structure permutation; (\* fixed \*)  
 $\tau \in S_k$  - an exposure order (\* currently fixed \*)  
OUTPUT: an embedding  $f_k : [1..k] \mapsto [1..n]$  of  $\pi$  in  $\sigma$ , or a message “NONE”;

```

set  $T_0 = \text{NULL}$ ;
for  $i = 1$  to  $k$  do {
    set  $T_i = \emptyset$  ;
    for each row of  $T_{i-1}$  (* corresponding to some  $f_{i-1}$  *) do {
        (* try to expand  $f_{i-1}$  to  $f_i$  *)
        for  $j = 1$  to  $n$  do {
            (* check whether  $j$  is legitimate value for  $f_i(\tau(i))$  *)
            check whether  $f_{i-1}(p_i^-) < j < f_{i-1}(p_i^+)$ ; (* Monotonicity *)
            check whether  $\sigma(f_{i-1}(q_i^-)) < \sigma(j) < \sigma(f_{i-1}(q_i^+))$ ;
            (* Order Preservation *)
            if both checks succeeded, do {
                create a new row of  $T_i$ ;
                fill the numerical fields suitably,
                using data from the current row of  $T_{i-1}$  and the value  $j$ ;
                direct the pointer to that row;
                if the new row did not occur yet in  $T_i$ , add it to  $T_i$ ;
            } } } }
if  $T_k$  is empty return “NONE”;
else, using pointers, reproduce an actual embedding  $f_k$ , and return it.
```

The correctness of the algorithm follows from the definition of the significant elements, and Proposition II. What is its time complexity? First, we need to compute all  $\{\partial_\pi A_i\}$ , which will take a time at most quadratic in  $k$ . Then, at every stage  $i$ ,  $O(n)$  operations are performed per each row of  $T_{i-1}$ . In addition, one must take care not to create identical rows in  $T_i$ . This can be done, e.g., by performing an on-line bucket sort, using a table of size  $O(n^{|\partial_\pi A_i|})$ . (Note that  $|\partial_\pi A_i| \leq 1 + |\partial_\pi A_{i-1}|$ .) Thus, the entire  $i$ -th stage can be implemented in time  $O(n^{1+|\partial_\pi A_{i-1}|})$ . Finally, the reconstruction of the resulting  $f_k$  may require an additional time  $O(k)$ . Altogether, we get

$$O(k^2) + O\left(\sum_{i=1}^k n^{1+|\partial_\pi A_{i-1}|}\right) = O\left(k^2 + kn^{1+\max_i |\partial_\pi A_i|}\right).$$

**Remark:** If our goal were to find all the embeddings  $\pi \hookrightarrow \sigma$ , instead of back-pointers we would use forward-pointers from  $f_{i-1}$  to *all* its extensions  $f_i$ ; notice that there are at most  $n$  such. The upper bound for the time it takes to fill up the tables would change by at most a multiplicative constant. Then, using a standard algorithm for DAG’s, we could find all the embeddings (corresponding to the paths from  $T_0$  to  $T_k$ ) in time proportional to that spent so far, plus  $k$  times the total number of such embeddings. ■

The parameter which governs the complexity of the algorithm is  $\max_i |\partial_\pi A_i|$ . Let us give this parameter a name. For a structure permutation  $\pi \in S_k$  and an exposure order  $\tau \in S_k$ , define  $C_\tau(\pi) = \max_{i \in [1..k]} |\partial_\pi A_i|$ . By the preceding discussion, the complexity of **ALG-SP** $_{\pi,\tau}$  is bounded by  $O(k^2 + kn^{1+C_\tau(\pi)})$ . Now, observe that the exposure order does not have to be fixed, and in fact it makes a good sense to choose the best possible  $\tau$  for the given  $\pi$ . Thus, we arrive at the key definition of this section:

**Definition 5.** Define  $C(\pi)$ , the complexity of a permutation  $\pi \in S_k$ , as

$$C(\pi) = \min_{\tau} C_\tau(\pi).$$

Our discussion so far can be summarized by the following theorem:

**Theorem 1.** The SP problem for a structure permutation  $\pi \in S_k$  can be solved in time

$$O(c_k + kn^{1+C(\pi)}),$$

where  $c_k$  is the time needed to compute the best  $\tau$ .

Alternatively, if finding the best  $\tau$  is too expensive, one can use a reasonably good  $\tilde{\tau}$  and get time complexity  $O(\tilde{c}_k + kn^{1+C_{\tilde{\tau}}(\pi)})$  where  $\tilde{c}_k$  is the time need to produce such  $\tilde{\tau}$ .

The remaining part of this paper is dedicated to the study of  $C(\pi)$ . In particular, we shall see how to produce  $\tilde{\tau}$  such that  $C(\pi) \leq C_{\tilde{\tau}}(\pi) = 0.35k + o(k)$ , indicate how a poly-log approximation of  $C(\pi)$  can be obtained in time polynomial in  $k$ , and discuss a number of concrete examples of  $\pi$ 's.

### 3 Complexity of Permutations

The graph  $G_\pi$  defined in the previous section provides a convenient way to approach many questions related to  $C(\pi)$ . Observe that in fact  $G_\pi$  is a multigraph obtained by taking a union of two Hamiltonian paths, blue and red. Conversely, any such multigraph corresponds to some  $\pi$  under the suitable labeling of vertices (defined by the blue path). The maximum degree of  $G_\pi$  is at most 4.

#### 3.1 General Upper Bounds

The first natural question to ask is how well can a *fixed* exposure order  $\tau$  perform. The answer is given by the following proposition:

**Proposition 3.** Let  $\pi$  be a structure permutation in  $S_k$ , and let the exposure order be *Id*, i.e.,  $(1, 2, 3, \dots, k)$ . Then,  $C_{Id}(\pi) \leq 2/3 \cdot k + 1$ . Conversely, for any fixed exposure order  $\tau$  there exists a structure permutation  $\pi$  such that  $C_\tau(\pi) \geq 2/3 \cdot k$ .

*Proof.* By the definition of  $G_\pi$ , the boundary  $\partial_\pi A_i$  is in fact a union of two boundaries: the blue one, created by the blue edges, and the red one, created by the red edges. In the case of exposure order  $Id$ , the blue boundary is always of size 1. Let us show that the size of the red boundary of *any* set  $A \subseteq V(G_\pi)$  is at most  $2/3 \cdot k$ . Since the red degree of any vertex  $v \in V(G_\pi)$  is at most 2 and at least 1, we conclude that  $|\partial_\pi^{red} A| \leq 2|\bar{A}|$ . Since  $|A| + |\bar{A}| = k$  and  $|\partial_\pi^{red} A| \leq |A|$ , the conclusion follows, yielding the first part of the theorem.

For the second part, let us first show that it is true for  $\tau = Id$ . It suffices to construct a permutation  $\pi \in S_k$  such that the  $\partial_\pi^{red} A_{2/3 \cdot k} = A_{2/3 \cdot k}$ . Here is an interesting concrete example of such a permutation: Let  $k = 3^m$ , and define  $\rho : [0..k - 1] \mapsto [0..k - 1]$  as a permutation which maps a number  $m$  in the range to the number whose trinary representation (with leading zeroes) is the reverse of the trinary representation of  $m$ . Since the image of  $[0..2/3 \cdot k - 1]$  consists of (all) numbers whose least significant digit is 0 or 1, while the image of  $[2/3 \cdot k..k - 1]$  consists of (all) numbers whose least significant digit is 2, and we see that every  $m \in [0..2/3 \cdot k - 1]$  has a red edge to  $[2/3 \cdot k..k - 1]$ . Thus,  $\rho$  has the desired property.

To complete the proof of the second part, observe that the structure permutation  $\pi = \rho \circ \tau^{-1}$  with respect to the exposure order  $\tau$  (which can be viewed as a permutation of the range) has red boundaries isomorphic to those of  $\rho$  with respect to  $Id$ . ■

Two remarks are due. First, we have treated so far  $\tau$  as the exposure order, which is of course a permutation of the range. We have refrained so far from calling  $\tau$  a permutation to avoid an unnecessary confusion. Second, as we shall see later in Section 5.3, there exists an exposing order  $\tau$  such that  $C_\tau(\rho) = O(\sqrt{k})$ . Thus, the identity permutation can be far off the best exposure order.

The next natural question is how well does a random exposure order behave.

**Theorem 2.** *For any structure permutation  $\pi \in S_k$ , almost all exposure orders  $\tau$  satisfy  $C_\tau(\pi) \leq (0.54 + o(1))k$ .*

*Proof.* The proving mechanism used here and in following theorems of this section is:

- Given a structure permutation  $\pi \in S_k$ , define a random process  $\mathcal{P}$  which produces the exposure order  $\tau \in S_k$  by exposing  $[1..k]$  vertex by vertex. (Between the consecutive exposures there can be some non-exposing activity.) The process  $\mathcal{P}$  will be described by specifying the conditional distribution according to which the next step is performed.
- For each  $i \leq k$  corresponding to the exposure of the  $i$ -th vertex, define the random variable  $X_i = |\partial_\pi(A_i)|$ , compute its expectation  $E[X_i]$ , and show that  $X_i$  is well concentrated, i.e., that for some function  $g(k) = o(k)$  it holds

$$\Pr [X_i - E[X_i] \geq g(k)] \leq o(1/k).$$

After having succeeded proving this property, we may conclude that a random permutation  $\tau$  drawn according to  $\mathcal{P}$  almost surely satisfies  $C_\tau(\pi) \leq \max_i E[X_i] + g(k)$ .

3. In order to prove the concentration property, the standard martingale technique is used (see, e.g., [2] and [6] for a detailed description of how the martingales are used in Discrete Mathematics). For each  $i$  and  $t = 0, 1, \dots, k$  we define a random variable  $Y_i^t$  as the expected value of  $X_i$  after the exposure of the first  $t$  vertices. Clearly,  $E[X_i] = Y_i^0$ , and  $\{Y_i^t\}_{t=0}^k$  form a martingale. I.e.,  $E[Y_i^{t+1} | Y_i^t] = Y_i^t$ . Then we establish the Bounded Differences Property, i.e., that there exists a constant  $L$  such that  $|Y_i^{t+1} - Y_i^t| \leq L$  for all  $i$  and  $t$ , and conclude, by Azuma Inequality, that

$$\Pr [X_i - E[X_i] \geq 2L\sqrt{k \ln k}] \leq 1/k^2.$$

Let us see how the above strategy works for analyzing random uniformly chosen  $\tau \in S_k$ . We define  $\mathcal{P}$  by saying that at each stage one of the unexposed vertices is drawn uniformly at random.

Let us first estimate  $E[X_i]$ . What is the probability of vertex  $v \in V(G)$  to belong to  $\partial_\pi A_i$ ? A simple combinatorial argument shows that if  $v$  has degree 4 (counting multiple edges as a single edge), then

$$\Pr [v \in \partial_\pi A_i] = \frac{i}{k} \left[ 1 - \frac{i-1}{k-1} \cdot \frac{i-2}{k-2} \cdot \frac{i-3}{k-3} \cdot \frac{i-4}{k-4} \right] \leq h(i/k) + o(1),$$

where  $h$  is a real valued function  $h(x) = x(1-x^4)$ . If the degree of  $v$  is less than 3, the above probability decreases. Thus, by linearity of expectation,

$$\max_i E[X_i] \leq k \max_{x \in [0,1]} h(x) + o(k) \leq 0.54k + o(k).$$

In order to complete the proof we have to verify that the exposure martingale  $\{Y_i^t\}_{t=0}^k$  has the Bounded Differences Property. But this is easy: since an exposure of an additional vertex can influence at most 4 other vertices, we get at once  $|Y_i^{t+1} - Y_i^t| \leq 1 + 4 = 5$ . ■

How can the constant 0.54 of Theorem 2 be improved? On one hand, the process  $\mathcal{P}$  should be more “sensitive” to the current boundary, and attempt not to increase it needlessly. On the other hand, if one wishes to follow the same general strategy,  $\mathcal{P}$  should be “stable” in the sense that the exposure of an extra vertex could have only a limited influence on the future behaviour of  $\mathcal{P}$ . Such behaviour is required to ensure the Bounded Differences Property, needed for the analysis of  $\mathcal{P}$ .

A natural improvement on the previous random process would be to fill up the “holes” (i.e., the unexposed vertices all of whose neighbours are exposed) upon their creation. Clearly, such an operation can only be beneficial for the size of the boundaries, and there is no gain in delaying it. We call the new process  $\mathcal{P}_1$ .

**Theorem 3.** *Almost all exposure orders  $\tau \in S_k$  produced by  $\mathcal{P}_1$  satisfy  $C_\tau(\pi) \leq k(0.46 + o(1))$ .*

*Proof. Sketch:* We follow closely the proof of Theorem 2. Using a similar, but much more involved technical analysis, we arrive at the conclusion that

$$\Pr[v \in \partial_\pi A_i] \leq h_1(|A_i|/k) + o(1),$$

where  $A_i$  is the set of all exposed vertices after the exposition of the  $i$ -th truly random (i.e., not forced by holes filling) vertex, and  $h_1$  is a real valued function determined by the local structure of  $G_\pi$ . It is found by a lengthy case analysis; e.g., in the typical case when  $G_\pi$  has (locally) degree 4, and no cycles of length  $\leq 4$ ,  $h(x) = \sum_{i=0}^4 \binom{4}{i} x^{5-i} (1-x)^i (1-x^{3i})$ . The maximum value of  $h_1(x)$  on  $[0, 1]$  is 0.46. Therefore, by linearity of expectation,

$$\max_i \mathbb{E}[X_i] \leq k \left( o(1) + \max_{x \in [0,1]} h(x) \right) \leq (0.46 + o(1)) k.$$

As before, in order to complete the proof it remains to check that the exposure martingale  $\{Y_i^t\}$  has the Bounded Differences Property. Although at the first glance one might suspect that the holes filling process might have a cascading effect, it actually cannot. It is not hard to get convinced that the exposure of an extra vertex at any particular moment can influence at most  $1 + 4 + 3 \cdot 4 = 17$  vertices (the vertex itself, its neighbours, and the neighbours' neighbours), and thus  $|Y_i^{t+1} - Y_i^t| \leq 17$ . We postpone a detailed explanation of this proof to the full version of the paper. ■

The process  $\mathcal{P}_1$  can be further improved with respect to the sizes of the boundaries. Besides holes filling, there is one more beneficial operation: if an exposure of a vertex does not increase the size of the current boundary, there is definitely no damage in exposing it. However, although we would like to perform this operation whenever possible, it may cause a cascading effect, destabilizing the process. In order to take care of this problem, for each  $d \geq 1$  we introduce an “approximating” stable process  $\mathcal{P}^d$ , defined as follows:

Every vertex will have a “height” in  $[1..d], \infty$ . The vertices with height  $< \infty$  will be precisely the exposed vertices. At each stage  $t$ , for each  $m = 1, \dots, d$ , let  $A_t^m$  be the set of all vertices whose height does not exceed  $m$ . As long as there exist vertices  $v$  of height  $> m + 1$ , whose addition to  $A_t^m$  does not increase the boundary of this set, pick randomly such a vertex  $v$ , and change its height to  $m + 1$ . If  $v$  was previously unexposed, it gets exposed during this operation. Now, if there are no such vertices, we pick a random vertex in  $[1..k]$  and change its height to 1 (exposing it if it was unexposed).

It can be shown (the details are postponed to the full version) that  $\mathcal{P}^d$  can have but a limited cascading, and that any random step may influence at most  $1 + 4^{d+1}$  vertices. Furthermore, the analysis of the expected size of the boundary can still be successfully carried out. (Although it becomes very messy). An analysis of  $\mathcal{P}^3$  yields the following theorem:

**Theorem 4.** *For every structure permutation  $\pi \in S_k$ ,  $\mathcal{P}^3$  produces with high probability an exposure order  $\tau$  such that  $C_\tau(\pi) \leq (0.35 + o(1))k$ . Hence, the complexity  $C(\pi)$  of  $\pi$  is at most  $(0.35 + o(1))k$ .*

Before concluding this section, we would like to mention an entirely different approach for determining an exposure order  $\tau$  for the given  $\pi$ . Take the Laplacian matrix of the graph, find its eigenvector  $e_1$  corresponding to the first positive eigenvalue, and expose the vertices in the increasing order of values of the entries of  $e_1$ . Numerical simulations seem to indicate that this method is superior to all other methods described in this section, leading to a constant about 0.22. There is also a good intuition why it should work (see, e.g., [7] for a related discussion). Unfortunately, we do not know how to prove this.

### 3.2 Approximation and a General Lower Bound

So far we have worked with the vertex boundary  $\partial A$  of the set  $A \in G$ . Let us introduce also the *edge boundary*  $DA$  of  $A \in G$ , defined by

$$DA = \{e \in E(G) \mid e \in E(A, \bar{A})\}.$$

Since our  $G$  has degree bounded by 4, it holds

$$|\partial_\pi A| \leq |D_\pi A| \leq 4|\partial_\pi A|. \quad (1)$$

The edge boundary of a set is one of the basic terms of Graph Theory, and its introduction permits to employ the existing machinery. Here is how it can be used to efficiently compute an approximation of  $C(\pi)$ :

**Theorem 5.** *An exposure order  $\tau$  for which  $C_\tau(\pi) \leq O(\log^2 k)C(\pi)$  can be constructed in time polynomial in  $k$ .*

*Proof.* Finding the order of vertex exposure  $\tau$  of  $V(G)$  for an (arbitrary) input graph  $G$  which minimizes the maximal  $DA_i$ , is a well known classical problem called the *minimum cut linear arrangement* problem. It is NP-complete. However, Leighton and Rao [5] have designed a polynomial  $O(\log^2 k)$  approximation for this problem. Using their algorithm on our graph  $G_\pi$ , and keeping in mind (II), we get the desired approximation. ■

Next, we prove that there exist permutations such that  $C(\pi) = \Omega(k)$ . In fact, almost all permutations have this property.

**Theorem 6.** *Let  $\pi \in S_k$  be a random, uniformly chosen permutation. Then there exists a universal constant  $c > 0$  such that*

$$\Pr[C(\pi) \leq ck] \leq o(1).$$

*Proof.* Assume for convenience that  $k$  is even. Call a graph  $G$   $\alpha$ -rich if any bisection of  $G$  (i.e., partition of  $V(G) = [1..k]$  to two equal parts) defines an edge-cut of size at least  $\alpha k$ . By (II), it suffices to show that a random  $G$  obtained

by taking a union of two randomly and independently chosen Hamiltonian paths  $P_1$  and  $P_2$  on the vertex set  $[1..k]$  is  $\alpha$ -rich for some constant  $\alpha > 0$ .

Denote by  $\mathcal{P}$  the uniform probability space over all the (multi-)graphs obtained by taking a union of two random Hamiltonian paths. We shall prove that there exist constants  $\alpha, \epsilon > 0$  such that for every bisection  $(V_1, V_2)$  the probability that a  $\mathcal{P}$ -random graph has a sparse (i.e.,  $\leq \alpha k$ ) edge-cut with respect to  $(V_1, V_2)$ , is less than  $2^{-k(1+\epsilon)}$ . That is, for every bisection  $(V_1, V_2)$ , it holds

$$\Pr_{\mathcal{P}}[|E_G \cap E(V_1, V_2)| \leq \alpha k] < 2^{-k(1+\epsilon)}. \quad (2)$$

Since there are  $2^{k-1} - 1$  bisections of the vertices  $[1..k]$ , the probability that there exists a “meager” bisection (i.e., one yielding an edge-cut of size  $\leq \alpha k$ ), is less than  $2^k 2^{-k(1+\epsilon)} = o(1)$ . Thus, almost surely, a  $\mathcal{P}$ -random graph is  $\alpha$ -rich.

It remains to establish  $\alpha, \epsilon$  for which Inequality 2 is true. Let  $(V_1, V_2)$  be a fixed bisection of  $[1..k]$ . Let us examine the distribution of the size of the edge cut defined by this bisection for a random  $G \in \mathcal{P}$ ; we shall call this random variable  $S$ .

For each path  $P_i$ ,  $i = 1, 2$ , define the indicator binary random variables  $\{X_{i,j}\}_{j=1}^k$ , which indicate whether  $P_i(j) \in V_1$  or  $P_i(j) \in V_2$ . We shall think of the set of these indicators  $\{X_{i,j}\}$  as of binary string  $X$  of length  $2k$  ordered in the following way:

$$X = (X_{1,1}, X_{1,2}, \dots, X_{1,k}; X_{2,1}, X_{2,2}, \dots, X_{2,k}).$$

Observe that the string  $X$  has a uniformly distribution over all binary strings of length  $2k$  with the property that both the first  $k$  bits and the last  $k$  bits are balanced (i.e., they contain  $k/2$  0's and  $k/2$  1's). Observe also that,

$$S = S(X) = \sum_{i=1,2; \quad 1 \leq j < k} X_{i,j} \oplus X_{i,j+1}.$$

where  $\oplus$  is the addition operation in  $Z_2$ .

A simple calculation shows that even after dropping the balancedness requirements, the number of binary strings  $Y$  of length  $2k$  for which  $S(Y) \leq \alpha k$  is

$$4 \sum_{t=0}^{\alpha k} \binom{2(k-1)}{t} \approx 2^{H(\alpha/2) \cdot 2k + o(k)},$$

were  $H(p) = -p \log_2 p - (1-p) \log_2 (1-p)$  is the Entropy function, and the standard approximation used is from e.g., [2]. Recalling that the string  $X$  is uniformly distributed over strings of length  $2k$  which are balanced with respect to both  $k$  first and  $k$  last bits, we conclude that

$$\Pr[S \leq \alpha k] \leq \frac{2^{H(\alpha/2) \cdot 2k + o(k)}}{2^{2k}/\Omega(k)} \leq 2^{-(1-H(\alpha/2)) \cdot 2k + o(k)}.$$

Taking  $\alpha$  such that  $H(\alpha/2) < 0.5$  yields Inequality 2, and completes the proof of the theorem. ■

### 3.3 Special Cases

In this section we consider two different types of “linear” permutations  $\pi \in S_k$ , and show that both types have complexity at most  $O(\sqrt{k})$ . Throughout this section, we think of permutations  $\gamma \in S_k$  as permutations of the set  $[0..k - 1]$ , and not of the usual set  $[1..k]$ .

The “linear” permutation  $\pi$  of the first type comes from an invertible linear transformation  $A : Z_2^m \mapsto Z_2^m$ . Let  $k = 2^m$ . The value  $\pi(i)$  of a number  $i \in [0..k - 1]$  is defined by taking the binary representation of  $i$  (with leading zeroes), interpreting it as a vector  $v \in Z_2^m$ , applying  $A$  to  $v$  to get  $u = A(v)$ , and interpreting  $u$  as the binary presentation of another number in  $[0..k - 1]$ .

For the second type, we consider invertible linear transformation of the ring  $Z_k$  to itself. Formally,

**Definition 6.** Let  $k = 2^m$  for some positive integer  $m$ . A permutation  $\pi \in S_k$  is called a linear permutation of the first type if it is an invertible linear transformation  $A_\pi : (Z_2)^m \mapsto (Z_2)^m$ , under identifying  $Z_2^m$  and  $[0..2^m - 1]$  by means of the binary representation.

A permutation  $\pi \in S_k$  is called a linear permutation of the second type if it is a permutation of the ring  $Z_k$  by an invertible linear transformation  $\pi(x) = ax + b \bmod k$ .

Before starting with bounding the complexity of linear permutations, let us state a useful property of the complexity function:

**Proposition 4.** Let  $\pi \in S_k$  be a structure permutation, and denote the identity permutation by  $Id$ . Then, for every  $S \subseteq [0..k - 1]$ ,  $|\partial_\pi(S)| \leq |\partial_{Id}(S)| + |\partial_{Id}(\pi(S))|$ . Consequently, for any  $\tau \in S_k$ ,  $C_\tau(\pi) \leq C_\tau(Id) + C_{\pi \circ \tau}(Id)$ .

*Proof.* The inequality holds since the first term captures the blue boundary, while the second term captures the red boundary. The consequence follows by considering the sets  $S_i = \tau[0..i]$ . ■

We start with showing that linear permutations of the first type have low complexity. Doing so, we show first that for a special subfamily of such permutations, the trivial exposure order  $\tau = Id$  achieves the desired  $O(\sqrt{k})$  bound.

**Theorem 7.** Let  $\pi \in S_k$  be a linear permutation of the first type, and let  $A_\pi$  be its matrix representation  $m \times m$ . If  $(A_\pi^{-1})_{i,j} = 0$  for every  $(i,j)$ -th entry,  $0 \leq i, j < m$ , with  $2j \leq i$ , then  $C_{Id}(\pi) \leq 2^{4+\lceil \frac{m}{2} \rceil}$ .

*Proof.* Let  $e_0, \dots, e_{m-1}$  be the vectors of the standard normal base of  $Z_2^m$ . We shall denote by  $V_r$  the linear subspace spanned by the vectors  $e_0, \dots, e_{r-1}$ . The addition operation in the  $Z_2^k$  vector space will be denoted by  $\oplus$ .

Assuming  $(A_\pi^{-1})_{i,j} = 0$  for every  $2j \leq i$ , we conclude that for every  $i \geq 0$  it holds  $V_i \supseteq A_\pi^{-1}V_{\lfloor \frac{i}{2} \rfloor}$ , and thus  $A_\pi V_i \supseteq V_{\lfloor \frac{i}{2} \rfloor}$ .

Interpreting the linear subspace  $V_i$  in terms of the original domain  $[0..2^m - 1]$ , we see that it is actually the subinterval  $[0..2^i - 1]$ . Thus, we have  $\pi([0..2^i - 1]) \supseteq$

$[0..2^{\lfloor \frac{i}{2} \rfloor} - 1]$ . Furthermore, since  $A_\pi V_i$  is a linear subspace of  $Z_2^k$  containing  $V_{\lfloor \frac{i}{2} \rfloor}$ , every vector  $u$  which is obtained from a vector  $v \in A_\pi V_i$  by arbitrarily modifying its first  $\lfloor \frac{i}{2} \rfloor$  coordinations also belongs to  $A_\pi V_i$ . This implies that  $\pi([2^i])$  is composed of subintervals, each one of length at least  $2^{\lfloor \frac{i}{2} \rfloor}$ . Keeping in mind that the blue boundary of a subinterval in  $G_{Id}$  contains at most two vertices, and that the maximal possible number of such subintervals in  $\pi([0..2^i - 1])$  is  $2^{\lceil \frac{i}{2} \rceil}$ , we conclude that

$$|\partial_\pi([0..2^i - 1])| \leq |\partial_{Id}([0..2^i - 1])| + |\partial_{Id}(\pi([0..2^i - 1]))| \leq 2 + 2 \times 2^{\lceil \frac{i}{2} \rceil} \leq 2^{\frac{i}{2}+2}.$$

In order to complete the proof we have to bound the size of the boundary  $\partial_\pi([0..t - 1])$  for every  $t < k$  (so far, we have taken care only of powers of two). Using similar arguments, it is easy to show that  $|\partial_\pi([2^i p..2^i p + 2^i - 1])| \leq 2^{\frac{i}{2}+2}$ . (Note that if  $x, y \in [2^i p, \dots, 2^i p + 2^i - 1]$  then  $x \oplus y \in [2^i]$  and thus  $(A_\pi(x) \oplus A_\pi(y)) \in V_{\lfloor \frac{i}{2} \rfloor}$ ). Now, every interval  $[0..t - 1]$  can be represented as the union of at most  $m$  subintervals of the form  $[2^i p, \dots, 2^i p + 2^i - 1]$ , where there is at most one subinterval of a given length. Combining the boundaries of those subintervals, we establish the bound  $|\partial_\pi([0..t - 1])| \leq \sum_{i=0}^{m-1} 2^{\frac{i}{2}+2} \leq 2^{\frac{i}{2}+4}$ . ■

We are ready to prove now the bound on the complexity of a general linear permutation of the first type.

**Theorem 8.** *For a linear permutation  $\pi \in S_k$  of the first type it holds  $C(\pi) = O(\sqrt{k})$ .*

*Proof.* Keeping in mind that, by Proposition 4,

$$C_\tau(\pi) \leq C_\tau(Id) + C_{\pi \circ \tau}(Id),$$

it suffices (in view of the previous theorem) to find  $\tau$  such that for every  $(i, j)$  with  $2j \leq i$ , the following requirements hold:  $(A_\tau^{-1})_{i,j} = 0$  and  $(A_{\pi \circ \tau}^{-1})_{i,j} = 0$ . Equivalently, we need

$$(A_\tau^{-1})_i e_j = 0 \quad \text{and} \quad (A_\tau^{-1})_i (A_\pi^{-1} e_j) = 0 \tag{3}$$

where  $(A_\tau^{-1})_i$  is the  $i$ -th row of  $A_\tau^{-1}$ . For every  $i$ , Equation 3 forces  $2 \lfloor i/2 \rfloor \leq i$  linear restrictions on the vector  $(A_\tau^{-1})_i$ . Thus, for every  $i$  there exist at least  $2^{m-i}$  vectors in  $(Z_2)^m$  which satisfy the restrictions of (3). Hence it is possible to construct the transformation  $A_\tau^{-1}$  in an inductive manner, starting with  $i = m-1$  and going down to  $i = 0$ , and calculating for each  $i$  the row  $(A_\tau^{-1})_i$ . Note that we have to come up with an invertible transformation, and thus at every stage we must make sure that the new vector  $(A_\tau^{-1})_i$  is linearly independent of the already constructed rows. Since we have  $2^{m-i}$  candidates for  $(A_\tau^{-1})_i$  satisfying the restrictions of (3), and the number of different linear combinations of previously constructed vectors is  $2^{m-i-1}$ , it is always possible to find a legal candidate which is independent of the old vectors. ■

We address now the linear permutations of the second type.

**Theorem 9.** *Let  $\pi$  be a linear permutation of the second type. Then, for some (explicitly constructed)  $\tau \in S_k$ , it hold  $C_\tau(\pi) \leq 4\sqrt{k}$ .*

*Proof.* Assume for simplicity that  $k$  is a square of an integer; the proof of the general case can be obtained along the similar lines.

Define the sequence  $\{x_i\}_{i=0}^{k\sqrt{k}-1}$  by

$$x_{t\sqrt{k}+r} = \pi^{-1}(r) + t \bmod k,$$

for every  $0 \leq t < k$  and  $0 \leq r < \sqrt{k}$ . Now, define the exposure order  $\tau$  by exposing the elements of  $Z_k$  in the order they appear in the sequence  $\{x_i\}$ . In the case the same value appears many times in  $\{x_i\}$ , we consider only the first appearance.

Observe that for every  $i \geq \sqrt{k}$  there exists  $j < i$  such that  $\tau(i)$  is a neighbour of  $\tau(j)$  in  $G_{Id}$  as  $x_{t\sqrt{k}+r} - x_{(t-1)\sqrt{k}+r} = 1$ . Therefore, it is possible to partition  $A_i$  into disjoint chains of neighbours, each “rooted” in the interval  $[0.. \sqrt{k}]$ . Thus,  $A_i$  is composed of at most  $\sqrt{k}$  intervals. Therefore,  $C_\tau(Id) \leq 2\sqrt{k}$ .

Next, we aim to bound  $C_{\pi \circ \tau}(Id)$ . Notice that for every  $t \geq 0$  and  $0 \leq r < \sqrt{k}$  the sequence  $\pi(x_{t\sqrt{k}}), \pi(x_{t\sqrt{k}+1}), \dots, \pi(x_{t\sqrt{k}+r-1})$  is a continuous interval, since

$$\pi(x_{t\sqrt{k}+r+1}) - \pi(x_{t\sqrt{k}+r}) = \pi(\pi^{-1}(r+1) + t) - \pi(\pi^{-1}(r) + t) = (r+1) - r = 1.$$

Thus,  $A_i^{\pi \circ \tau}$  is also composed of at most  $\sqrt{k}$  intervals, which implies  $C_{\pi \circ \tau}(Id) \leq 2\sqrt{k}$ . Consequently, by Proposition 4,  $C_\tau(\pi) \leq C_\tau(Id) + C_{\pi \circ \tau}(Id) \leq 4\sqrt{k}$ . ■

## References

1. Alon N., Friedgut E., On the number of permutations avoiding a given pattern. JCTS, in press.
2. Alon N., Spencer J. H., Erdos P., *The Probabilistic Method*, Wiley-Interscience, pp. 83-93, 1991.
3. Bona M., Exact enumeration of 1342-avoiding permutations; A close link with labeled trees and planar maps, *Journal of Combinatorial Theory, Series A*, 80, pp. 257-272, 1997.
4. Bose P., Buss J. F., Lubiwi A., Pattern matching for permutations, *Proceeding of the 3rd Workshop on Algorithms and Data Structures*, 1993.
5. Leighton F.T., Rao S., An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In Proceeding of 29-th FOCS, 1988, pp.422-431.
6. McDiarmid C., Concentration. In *Probabilistic methods for Algorithms and Discrete Mathematics*, Ed. Habib et al., Springer 1998. pp. 195-248.
7. Spielman A. D., Teng S., Spectral partitioning works: planar graphs and finite element meshes, *Proceeding of the 37th Annual IEEE Conference on Foundations of Computer Science*, 1996.

# Parallel Attribute-Efficient Learning of Monotone Boolean Functions

Peter Damaschke

FernUniversität, Theoretische Informatik II  
58084 Hagen, Germany  
`Peter.Damaschke@fernuni-hagen.de`

**Abstract.** We consider exact learning of monotone Boolean functions by membership queries, in the case that only  $r$  of the  $n$  variables are relevant. The learner proceeds in a number of rounds. In each round he submits to the function oracle a set of queries which may be chosen depending on the results from previous rounds. In a STOC'98 paper we proved that  $O(2^r + r \log n)$  queries in  $O(r)$  rounds are sufficient. While the query bound is optimal for trivial information-theoretic reasons, it was open whether parallelism can be improved without increasing the amount of queries. In the present paper we prove a negative answer:  $\Theta(r)$  rounds are necessary in the worst case, even for learning a very special type of monotone function. The proof is an adversary argument, based on a distance inequality in binary codes. On the other hand, a Las Vegas strategy based on another STOC'98 result can learn monotone functions in  $2 \log_2 r + O(1)$  rounds, without using significantly more queries. We also study the constant factors in the deterministic case.

## 1 Introduction and Contributions

In the widely known model of *exact learning by membership queries*, a Boolean function  $f$  on  $n$  variables is given as an oracle (“black box”), and a learner wants to identify  $f$ . To this end he may ask queries of the following type: He chooses an assignment (giving Boolean value 0 or 1 to each variable), and the oracle provides the value of  $f$  for this assignment. Trivially, all  $2^n$  possible queries must be asked if nothing about  $f$  is known in advance. However, clever query strategies can exist if it is promised to the learner that  $f$  belongs to some restricted class of Boolean functions. There are trivial classes where still  $2^n$  queries are necessary, e.g. the class of functions that have value 1 for exactly one assignment. Even randomization cannot help in such cases. (Due to some recent fascinating results, quantum computers can solve search problems by surprisingly few queries, subject to some error probability; see e.g. the survey article [19]. But in the present paper we remain in the classical setting.)

Here we are concerned with a function class which can be efficiently learned by membership queries, namely monotone Boolean function where at most  $r$  of the  $n$  variables are relevant. A Boolean function  $f$  is *monotone* if  $\forall i : x_i \leq y_i$  implies  $f(x_1, \dots, x_n) \leq f(y_1, \dots, y_n)$ . A variable is *irrelevant* if switching its value

in arbitrary assignments does never change the function value. Otherwise it is a *relevant* variable. In the literature, the term *attribute-efficient learning* refers to learning strategies whose complexity is bounded by certain functions in both  $n$  and  $r$ , usually by  $\log n$  and a polynomial in the length of a representation (which may be  $2^r$ ).

We consider *learning in rounds*. In each round, the learner chooses a set of queries (assignments), and sends all these queries in parallel to the oracle. Then he may perform any computations with the obtained function values. In particular, the choice of the query set for each round may depend on the results obtained in previous rounds. In a randomized strategy, it may also depend on random bits. Learning in one round is called *nonadaptive learning*.

Various aspects of attribute-efficient learning have been studied in [3,4,10], [13,14,16,18,20,21]; this list is not exhaustive. An important special case of attribute-efficient learning is group testing, i.e. function  $f$  is known to be the disjunction of the relevant variables; see e.g. [7,8,9,12,15]. Group testing, as well as attribute-efficient learning in general, has interesting applications in fields like chemical and biological test series, error search in hardware and software, and pattern recognition; we refer to the several pointers in the above mentioned papers. Parallelity is essential to applications where the tests (queries) are time-consuming but can be executed simultaneously [12,11].

In [5] we devised a strategy that learns monotone functions with  $r$  relevant variables, using a total of  $O(2^r + r \log n)$  queries in  $O(r)$  rounds. It should be noticed that the learner is not assumed to know  $r$  in advance. The query bound is optimal for trivial information-theoretic reasons. In the present paper we prove that the number of rounds is also optimal, in the following sense: For deterministic strategies, there is an exponential tradeoff between the number of rounds and the coefficient of  $\log_2 n$  in the query number. Consequently, any strategy that uses an optimum number of queries needs  $\Theta(r)$  rounds in the worst case. The result is even true for a specific monotone Boolean function whose structure may be told the learner; only the location of the relevant variables must be kept secret. Our proof is an adversary argument, using an inequality for Hamming distances in binary codes. Due to this lower bound result, it makes sense to study the constant factors in nearly query-optimal strategies. Refining our strategy from [5], we obtain some concrete constants.

On the other hand, we give a randomized strategy that learns monotone functions by  $2^{O(r)} + O(r \log n)$  expected queries in only  $2 \log_2 r + O(1)$  expected rounds. (In fact, this remains true for more general function classes.) It is based on another result from [5] on nonadaptive attribute-efficient learning of arbitrary Boolean functions and uses standard ideas from hashing and the doubling technique.

## 2 Lower Bound for Queries vs. Rounds

Recall that the *Hamming distance* of two bit vectors is the number of positions where the vectors disagree. It is a straightforward exercise to prove that among

any three binary words of equal length there exist two words whose Hamming distance is at most  $\frac{2}{3}$  the length. This already enables us to prove a weaker version of Lemma 1 below. However, a stronger estimate on the minimum Hamming distance  $d$  between  $n$  binary code words of length  $q$  is known, namely:  $n \leq \frac{2d}{2d-q}$ . This is the *Plotkin bound*; see [17] or any textbook on error-correcting codes. It follows  $d \leq \frac{1}{2} \frac{n}{n-1} q$ .

**Lemma 1.** *There exists an adversary strategy such that the learner can identify at most  $b \log_2 s$  relevant variables in each round of  $s \log_2 n$  queries, where  $\lim_{n \rightarrow \infty} b = 1$ .*

*Proof.* The adversary chooses the following monotone Boolean function  $f$ . The  $r$  relevant variables of  $f$  are indexed  $x_1, x_2 \dots x_r$ , and the DNF contains all terms consisting of an  $x_j$ ,  $j$  odd, along with all  $x_i$ ,  $i < j$ ,  $i$  even. That means,  $f$  can be written as

$$x_1 \vee x_2 x_3 \vee x_2 x_4 x_5 \vee x_2 x_4 x_6 x_7 \vee x_2 x_4 x_6 x_8 x_9 \vee \dots$$

The adversary may even betray that the function is of this type, but he does not tell the learner what are the relevant variables.

We discuss the crucial property of this function:  $x_1 = x_2$  implies that  $f$  has the same value as  $x_1$ . Moreover,  $x_1 = 1$  implies  $f = 1$ . By the self-similar structure of this function, this repeats with later variables: If  $j$  is odd,  $x_i = 0$  for all odd  $i < j$ , and  $x_i = 1$  for all even  $i < j$ , then  $x_j = x_{j+1}$  implies that  $f$  gets the same value as  $x_j$ .

Assume that the learner asks  $s \log_2 n$  queries in the first round. The query set can be considered as a binary code of length  $q = s \log_2 n$ , assigning a code word to each variable. Exploiting the Plotkin bound, the adversary chooses two words (i.e. variables)  $y, z$  which differ in at most  $\frac{s}{2} \log_2 n$  bits, and settles that  $\{y, z\}$  is the pair of variables  $\{x_1, x_2\}$ . (Actually, as seen above, the guaranteed fraction is slightly larger than  $\frac{1}{2}$ , but the error tends to 0 with  $n \rightarrow \infty$ . For notational convenience we pretend henceforth that half of the bits are different.) Next we describe how the adversary answers the queries of the round. For each query where  $x_1$  and  $x_2$  get equal values, of course, the adversary outputs 1 if both  $x_1$  and  $x_2$  are 1 there, and 0 otherwise. The point is that these queries do not provide any information about the remaining relevant variables: No matter which variables are the  $x_i$ ,  $i \geq 3$ , the adversary's answers are consistent. In other words, the learner can exploit at most the other  $\frac{s}{2} \log_2 n$  queries, in order to recognize further relevant variables. Moreover, the latter subset of queries can be split in two subsets, one with  $y = 1, z = 0$ , and one with  $y = 0, z = 1$ . If the former subset is the majority then the adversary will fix  $x_1 = y, x_2 = z$ , and vice versa in the other case. Similarly as above, queries with  $x_1 = 1$  are answered by 1 and provide no information about further relevant variables. Thus, even worse, the learner can exploit at most  $\frac{s}{4} \log_2 n$  queries (where  $x_1 = 0, x_2 = 1$ ) to recognize further relevant variables.

The adversary considers this subset of queries again as a (shorter) binary code, and chooses two variables of minimum Hamming distance to be  $x_3$  and  $x_4$ ,

and so on. Then the above argument can be repeated. After  $\log_4 s$  steps, there remains a code of length  $\log_2 n$  or less, hence the adversary may even choose two of the  $n$  variables which have not been distinguished by the remaining queries. We conclude that the learner can identify only the first  $2 \log_4 s = \log_2 s$  relevant variables in the first round, without having gained further information on the other ones.

Clearly, this argument also applies to the following rounds, always starting with the earliest  $x_j$  not recognized yet. •

Now it is easy to derive the following lower bound.

**Theorem 1.** *Any deterministic strategy that learns monotone Boolean functions with  $r$  relevant variables in  $k$  rounds needs at least  $kc^{r/k} \log_2 n$  queries in the worst case, where  $\lim_{n \rightarrow \infty} c = 2$ .*

*Proof.* Let  $r_i$  be the number of relevant variables learned in the  $i$ th round, provided that the adversary observes the strategy of Lemma 11. Then we know that the  $i$ th round consists of at least  $2^{r_i/b} \log_2 n$  queries. Since  $\sum_i r_i = r$  and  $2^{t/b}$  is a convex function in  $t$ , the lower bound for the total query number  $\sum_i 2^{r_i/b} \log_2 n$  is minimized if  $r_i = r/k$  for all  $i$ . Thus the result follows with  $c = 2^{1/b} \rightarrow 2$ . •

**Corollary 1.** *Any deterministic strategy using (some constant factor within) the optimum number of queries can be forced to spend  $\Omega(r)$  rounds.* •

Theorem 1 gives a lower bound for the queries vs. rounds tradeoff. It is an open problem whether this bound is tight. In other words: Does there exist a learning strategy for monotone Boolean functions which needs only  $kc^{r/k} \log_2 n$  queries, for any number  $k$  of rounds between 1 and  $\Theta(r)$ ? The difficulty is to recognize some proper subset of relevant variables by a restricted number of nonadaptive queries. The presence of further relevant variables and an unlucky choice of queries may obscure their relevance, cf. [79].

### 3 Randomization Helps

In this section we show that a randomized strategy can achieve both coefficient  $r$  in the query number and much less than  $r$  rounds. We have already applied a similar technique in [6] for a different problem related to attribute-efficient learning. (See also [7] for a randomized solution of a search problem which provably does not allow an efficient deterministic strategy.)

As a preparation we recall the main result of [5], which has a proof of several pages:

**Theorem 2.** *Arbitrary Boolean functions with at most  $s$  relevant variables can be learned by  $O(s^2 2^s + s 2^s \log n)$  queries in one round, if  $s$  is known in advance.*

•

Using this upper bound, we can prove a result which contrasts nicely to the deterministic case. Before this, we explain the notion of *coarsening* of a Boolean function  $f$ : Assume that the set of variables is partitioned into so-called bins. The coarsening  $g$  of  $f$  with respect to this partition is defined as follows. The variables of  $g$  are the bins, and for any assignment of Boolean values to the bins, the value of  $g$  equals the value of  $f$  if each variable inherits the value assigned to the bin it belongs to. In particular, an oracle for  $f$  can be used as an oracle for  $g$ . Empty bins (containing no variables of  $f$ ) are considered as irrelevant.

Note that, in the following strategy, the learner is *not* assumed to have prior knowledge about  $r$ .

**Theorem 3.** *There is a Las Vegas strategy for learning monotone Boolean functions using  $2^{O(r)} + O(r \log n)$  expected queries in  $2 \log_2 r + O(1)$  expected rounds, if  $r$  of the  $n$  variables are relevant.*

*Proof.* For  $s = 1, 2, 4, 8, 16, \dots$  perform 3 rounds as described below, until the termination criterion in (3) is fulfilled:

(1) Throw the  $n$  variables at random into  $2^{3s}$  bins. Then consider the induced coarsening  $g$  of  $f$ , and apply a strategy due to Theorem 2 to learn  $s$  relevant bins by  $O(s^2 2^s)$  queries. Note that this step fails if still  $s < r$  and if  $g$  should have more than  $s$  relevant bins.

(2) Search for one relevant variable in each relevant bin. If a bin contains exactly one relevant variable, this can be done by  $\log_2 n$  queries in one round, and for all such bins in parallel. (Details need some care, but are quite obvious.) This is a total of at most  $s \log_2 n$  queries.

(3) Test whether all relevant variables have been found, otherwise double  $s$  and repeat. For this termination test, try all possible assignments  $y$  on the detected relevant variables and assign 0 to all remaining variables. Similarly, assign 1 to all remaining variables. These are at most  $2^{s+1}$  queries. Due to monotonicity of  $f$ , no further relevant variables exist if and only if, for every  $y$ , the all-0 and the all-1 assignment give the same function value.

Although the first rounds may fail, (3) ensures correctness of the final outcome. Consider the triples of rounds when  $s \geq r$  is already reached. These triples are called trials in the following. As soon as the  $r$  relevant variables get into  $r$  different bins in a trial, all relevant bins will now be found in (1), and all the relevant variables will be found in (2), which is then verified in (3). Hence a failure in a trial appears only if the relevant variables are not thrown into  $r$  distinct bins. Therefore the failure probability is at most  $r^2/2^{3s}$ .

The first trial needs  $2^{O(r)}$  queries in (1) and (3), and  $O(r \log n)$  queries in (2). Since  $s$  is always doubled, the query number in all previous rounds (with  $s < r$ ) is within these bounds. After a failure in a trial with parameter  $s$ , the next trial, with parameter  $2s$ , will ask  $O(4s^2 2^{2s})$  queries in (1) and (3), and  $2s \log_2 n$  queries in (3). However it is performed with probability less than  $r^2/2^{3s}$ . (Actually, this is a generous bound.) That means, the expected query number is bounded by  $O(r^2 s^2 / 2^s + r^2 s (\log n) / 2^{3s})$ . The sum of these terms over all trials is convergent,

hence the first trial dominates the expected query number. The expected number of rounds is obviously  $2 \log_2 r + O(1)$ . •

We conclude this section with a number of remarks.

(i) If  $r$  is known prior to learning then, obviously, the same query number can be achieved in  $O(1)$  rounds. The only critical point is to guess  $r$ .

(ii) We have used  $2^{3s}$  bins for ease of presentation only; a smaller number of bins would be sufficient. Moreover we may simplify the strategy: If the prospective number of bins should exceed  $n$ , we may consider  $f$  instead of a coarsening, thus the failure probability is 0.

(iii) Note that monotonicity is not really exploited. We used it only in (3), to test for the existence of further relevant variables. The same strategy is applicable to either class of Boolean functions which has the following properties: The class is closed under projection (i.e. fixing partial assignments), and it admits an  $O(1)$  query test, deciding whether a function from the class is constant.

(iv) The use of Theorem 2 is essential to our strategy. We do not see how to avoid it.

(v) Our proof in [5] shows that a family of  $O(s^2 2^s + s 2^s \log n)$  random assignments is sufficient for identifying functions with at most  $s$  relevant variables, with high probability. Since our strategy is highly random anyway, we may use random assignments in (1), instead of explicitly constructed families (which is apparently a very difficult matter).

(vi) Another important issue besides pure query complexity is the computational complexity, i.e. the amount of auxiliary computation to identify  $f$  from the answers given by the oracle. The only problem lies in the application of Theorem 2. For satisfactory solutions we refer to [6]. In particular, we can derive time bounds like  $O(\exp(r) + n \log n)$ , which resembles to the notion of fixed-parameter tractability.

## 4 The Coefficients in Nearly Query-Optimal Deterministic Strategies

In view of Corollary 11, it is now worthy to study the constant factors  $u, v, w$  in deterministic strategies using  $u2^r + vr \log_2 n$  queries in  $wr$  rounds.

First we observe  $u \geq 2$ : Even if the learner gets the relevant variables for free,  $2^{r+1}$  queries are necessary to verify that these are in fact the only relevant variables. Namely, for each assignment on the alleged relevant variables, two queries must be asked, where all other variables are 0 or 1, respectively. On the other hand, a total of  $2^{r+1}$  queries during the whole learning process (that is  $u = 2$ ) are sufficient for testing whether all relevant variables have been found, due to a simple argument: Even if some termination tests indicate further relevant variables, the learner does not have to repeat earlier queries of this type. Altogether,  $u$  is the least interesting constant. For  $v$  we have the trivial information-theoretic lower bound  $v \geq 1$ . It arises the question how small  $v$  can be actually made. The following strategy has  $v = 2 + o(1)$ ,  $w = 3$ , and  $u = 2$ . The skeleton of

this strategy has been given in [5], but there we did not worry about constant factors, therefore some refinements of the strategy itself and of the analysis are necessary.

We use some loose but convenient notion: An assignment  $x$  is identified with the set  $W$  of variables having value 1 in  $x$ . Consequently, we write  $f(W)$  for  $f(x)$ , and to “query a set  $W$ ” means to ask  $f(W)$ .

**Theorem 4.** *Monotone Boolean functions with at most  $r$  relevant variables can be learned by  $2^{r+1} + (2 + o(1))r \log_2 n$  queries in  $3r$  rounds.*

*Proof.* Let  $f$  be the given function and  $V$  its set of  $n$  Boolean variables. Assume  $f(\emptyset) = 0$  and  $f(V) = 1$ , otherwise  $f$  is constant, and we are done.

We arrange the variables as a  $q$ -hypercube of dimension  $q = \lceil \log_2 n \rceil$ , some irrelevant dummy variables may be added if  $n$  is not a power of 2. This naturally defines  $q$  pairs of  $(q-1)$ -hypercubes. In the first round we query  $q$  of these  $(q-1)$ -hypercubes, exactly one from each pair. If  $W$  has been queried and  $f(W) = 0$  then  $W$  becomes a so-called *lower set*, and  $V \setminus W$  becomes an *upper set*. Similarly, if  $f(W) = 1$  then  $W$  is declared to be an upper set, and  $V \setminus W$  is a lower set.

There follow two rounds with  $q$  assignments formed in the following way (but not all of them will be queried). Arrange the upper sets in arbitrary order. Then the candidate sets to be queried are the intersections of the first  $i$  upper sets, for  $i = 1, \dots, q$ . We partition this chain of nested sets into approximately  $\sqrt{q}$  segments of length about  $\sqrt{q}$ . In the second round, we query the first set (containing a single variable  $v$ ) and the  $\sqrt{q}$  sets bounding the segments. If  $f(\{v\}) = 1$  then  $v$  is relevant. Hence we have found some relevant variable after 2 rounds with  $q + \sqrt{q}$  queries.

Consider the other case  $f(\{v\}) = 0$ . There must be a jump from  $f = 0$  to  $f = 1$  in our chain. In the third round we query the  $\sqrt{q}$  sets of the segment where this jump occurred. Let  $W$  and  $W \cup W'$  be those neighbored sets in our chain with  $f(W) = 0$  and  $f(W \cup W') = 1$ . By construction, there is a unique pair of  $(q-1)$ -hypercubes  $H$  and  $H'$  such that:  $W \subset H$ ,  $W' \subset H'$ ,  $H$  is an upper set, and  $H'$  is a lower set.

We have two subcases. If  $f(H) = f(H')$  then one easily sees that both  $H$  and  $H'$  contain at least one relevant variable. (Since  $f$  is monotone, this holds for any complementary pair of sets.) If  $f(H) \neq f(H')$  then obviously  $f(H') = 0$ . Since  $f$  is monotone, this implies  $f(W') = 0 = f(W)$ . Together with  $f(W \cup W') = 1$ , we conclude as above that both  $W$  and  $W'$  contain relevant variables.

In order to distinguish these subcases, the learner has to query both  $H$  and  $H'$ . In the first round, only one of them has been queried. The second query might be asked in a fourth round, but we can save this round by asking all these partner queries in the interesting segment already in the third round. These are  $\sqrt{q}$  additional queries. In summary, we have found two disjoint subsets containing relevant variables, after 3 rounds with  $q + 3\sqrt{q}$  queries. This situation is called a *splitting*.

The whole process described above is recursively applied on the mentioned subsets, while the suitable assignment on the remaining variables is fixed. (De-

tails should be clear.) This yields a binary tree whose nodes are subsets of variables, and whose leaves contain one detected relevant variable each. If no further splitting is made then the tree is ready, and we test whether all relevant variables already appear as leaves. This termination test was described prior to the theorem, and we argued that  $2^{r+1}$  queries are needed during the whole search. In the negative case we find some non-constant “projection” of  $f$  and can continue the process with a new tree.

Altogether we obtain a sequence of binary trees. The total number of inner nodes is at most  $r - 1$ , and the sum of depths is at most  $r$ . An inner node represents 3 rounds, a leaf represents 2 rounds, but the termination tests add a further round to the deepest leaves in each tree. Hence every node represents 3 rounds, which gives  $w = 3$  in the worst case. The number of queries (excluding those of the termination tests) is bounded by

$$r(q + \sqrt{q}) + (r - 1)(q + 3\sqrt{q}) = (2r - 1)q + (4r - 3)\sqrt{q} = (2 + o(1))q.$$

(For  $\log n > 16r^2$  we even have  $v \leq 2$ .) •

Once more, there remain some open questions. First, we conjecture that  $v = 2$  is optimal. Furthermore, we do not know any positive constant lower bound for  $w$ . We only have an alternative strategy with  $w = 2$ , but  $v = 6$ , and  $u = r$ . (Here  $u$  is no longer constant, since we perform termination tests for many variables which are only suspected to be relevant.) But it is not clear whether  $w$  can be made arbitrarily small, at cost of large constant  $v$ . The same adversary construction as in Lemma 1 gives some lower bound for the  $v - w$ -tradeoff, but we have no strategy which achieves the corresponding *upper* bound. The problem is of similar nature as that mentioned at the end of Section 2.

## References

1. D.J. Balding, D.C. Torney: A comparative survey of non-adaptive pooling designs, in: *Genetic Mapping and DNA Sequencing (IMA Volumes in Mathematics and Its Applications)* (Springer 1995), 133-155
2. D.J. Balding, D.C. Torney: Optimal pooling designs with error detection, *Journal of Comb. Theory A* 74 (1996), 131-140
3. A. Blum, L. Hellerstein, N. Littlestone: Learning in the presence of finitely or infinitely many irrelevant attributes, *Journal of Computer and System Sciences* 50 (1995), 32-40
4. N.H. Bshouty, L. Hellerstein: Attribute-efficient learning in query and mistake-bound models, *9th Conf. on Computational Learning Theory COLT'96*, 235-243
5. P. Damaschke: Adaptive versus nonadaptive attribute-efficient learning, *30th ACM Symp. on Theory of Computing STOC'98*, 590-596, accepted by *Machine Learning*
6. P. Damaschke: Computational aspects of parallel attribute-efficient learning, *9th Int. Workshop on Algorithmic Learning Theory ALT'98, LNAI 1501*, 103-111
7. P. Damaschke: Randomized group testing for mutually obscuring defectives, *Info. Proc. Letters* 67 (1998), 131-135

8. A. De Bonis, L. Gargano, U. Vaccaro: Group testing with unreliable tests, *Info. Sci.* 96 (1997), 1-14
9. A. De Bonis, U. Vaccaro: Improved algorithms for group testing with inhibitors, *Info. Proc. Letters* 67 (1998), 57-64
10. A. Dhagat, L. Hellerstein: PAC learning with irrelevant attributes, *35th IEEE Symp. on Foundations of Computer Science FOCS'94*, 64-74
11. M. Farach, S. Kannan, E. Knill, S. Muthukrishnan: Group testing problems in experimental molecular biology, *Compression and Complexity of Sequences '97*, 357-367
12. P. Fischer, N. Klasner, I. Wegener: On the cut-off point for combinatorial group testing, *Discrete Applied Math.* 91 (1999), 83-92
13. T. Hofmeister: An application of codes to attribute-efficient learning, *5th European Conf. on Computational Learning Theory EuroCOLT'99, LNAI 1572* (1999), 101-110
14. J. Kivinen, H. Mannila, E. Ukkonen: Learning hierarchical rule sets, *5th Conf. on Computational Learning Theory COLT'92*, 37-44
15. E. Knill: Lower bounds for identifying subset members with subset queries, *6th ACM-SIAM Symp. on Discrete Algorithms SODA'95*, 369-377
16. N. Littlestone: Learning quickly when irrelevant attributes abound: a new linear-threshold algorithm, *Machine Learning* 2 (1988), 285-318
17. M. Plotkin: Binary codes with specified minimum distances, *IEEE Trans. Info. Theory* 6 (1960), 445-450
18. R.A. Servedio: Computational sample complexity and attribute-efficient learning, *31st ACM Symp. on Theory of Computing STOC'99*
19. A. Ta-Shma: Classical versus quantum communication complexity, *SIGACT News* 30(3) (1999), 25-34
20. R. Uehara, K. Tsuchida, I. Wegener: Optimal attribute-efficient learning of disjunction, parity, and threshold functions, *3rd European Conf. on Computational Learning Theory EuroCOLT'97, LNAI 1208*, 171-184
21. L.G. Valiant: Projection learning, *11th Conf. on Computational Learning Theory COLT'98*, 287-293

# Max- and Min-Neighborhood Monopolies

Kazuhisa Makino<sup>1</sup>, Masafumi Yamashita<sup>2</sup>, and Tiko Kameda<sup>2</sup>

<sup>1</sup> Division of Systems Science, Graduate School of Engineering Science, Osaka University, Toyonaka, Osaka, 560-8531, Japan. [makino@sys.es.osaka-u.ac.jp](mailto:makino@sys.es.osaka-u.ac.jp)

<sup>2</sup> Department of Computer Science and Communication Engineering, Kyushu University, Hakozaki, Higashi-ku, Fukuoka, 812-8581, Japan.  
[mak@csce.kyushu-u.ac.jp](mailto:mak@csce.kyushu-u.ac.jp)

<sup>3</sup> School of Computing Science, Faculty of Applied Sciences, Simon Fraser University, Burnaby, British Columbia, V5A 1S6 Canada. [tiko@cs.sfu.ca](mailto:tiko@cs.sfu.ca)

**Abstract.** Given a graph  $G = (V, E)$  and a set of vertices  $M \subseteq V$ , a vertex  $v \in V$  is said to be *controlled* by  $M$  if the majority of  $v$ 's neighbors (including itself) belongs to  $M$ .  $M$  is called a *monopoly* if every vertex  $v \in V$  is controlled by  $M$ . For a specified  $M$  and a range for  $E$  ( $E_1 \subseteq E \subseteq E_2$ ), we try to determine  $E$  such that  $M$  is a monopoly in  $G = (V, E)$ . We first present a polynomial algorithm for testing if such an  $E$  exists, by formulating it as a network flow problem. Assuming that a solution  $E$  does exist, we then show that a solution with the maximum or minimum  $|E|$  can be found in polynomial time, by considering them as weighted matching problems.

In case there is no solution  $E$ , we want to maximize the number of vertices controlled by the given  $M$ . Unfortunately, this problem turns out to be NP-hard. We therefore design a simple approximation algorithm which guarantees an approximation ratio of 2.

## 1 Preliminary

Let  $G = (V, E)$  be an undirected graph, where  $V$  (resp.,  $E$ ) is the vertex (resp., edge) set. We assume that  $G$  is simple, i.e.,  $G$  contains neither self-loops nor parallel edges. For a vertex  $v \in V$ , let us define the *neighborhood* of  $v$  by  $N_G(v) = \{v\} \cup \{w \mid (w, v) \in E\}$ . A vertex  $v \in V$  is said to be *controlled* by a vertex set  $M \subseteq V$  if the majority of its neighbors is in  $M$ , i.e.,

$$|N_G(v) \cap M| \geq |N_G(v)|/2. \quad (1)$$

Here we use a non-strict majority (including equality), however, all results obtained in this paper hold for the strict majority as well. For a vertex set  $M \subseteq V$ , let  $Cont(G, M)$  denote the set of vertices of  $G$  controlled by  $M$ . We call  $M$  a *monopoly* if it controls every vertex in the graph  $G$ , i.e.,  $Cont(G, M) = V$ .

The notion of monopoly was introduced by N. Linial et al. [8] to understand local majority voting in distributed computing. Local majority voting is motivated, for example, by agreement problems in agent systems (e.g., [2, 14, 15]). Let us consider the problem for the agents to agree on a standard from among

some proposals [14]. Under the assumption that every agent knows who supports which proposal, the agreement can be made simply by, for example, taking the one that the majority of the agents supports. The agent system, however, could be distributed too widely to admit of this solution, and this is why they suggested heuristic algorithms based on partial information on the distribution of the agents' votes.

For simplicity, suppose that there are two proposals, 0 and 1, and that the proposal that the majority of the agents supports is to be selected as the standard. Given, for each agent, the group of neighboring agents whose opinions are available to it, a simple and natural heuristic to approximate the agreement would be to take the majority of the opinions available to it, i.e., the opinions of its neighbors and itself. This is called the *deterministic local majority polling system*.

Peleg and his colleagues recently investigated such a system and determined how many agents supporting, say 0, are necessary and sufficient for the agreement to result in 0 [18, 12]. They model the system by an undirected graph  $G = (V, E)$ , where  $V$  and  $E$  respectively represent the set of agents and the (symmetric) neighborhood relation. Now, we can easily see that all agents decide on 0 if and only if there is a monopoly  $M$  in  $G$  whose members all support 0. In the deterministic local majority polling systems, ruling a monopoly  $M$  implies ruling  $V$  (i.e., all agents), and therefore, monopolies play an important role in such systems. Some other applications are discussed in [12].

Linial et al. [8] discussed the problems related to monopolies as packing and covering problems on graphs. They showed that  $|M|$  is  $\Omega(\sqrt{n})$  and gave a graph with a monopoly  $M$  of size  $O(\sqrt{n})$ , where  $n = |V|$ . As for computational complexity, Peleg [12] showed that the problem of computing a minimum monopoly is NP-hard, by reducing the minimum dominating set problem to it. Based on the non-approximable results [3, 9] on the set cover problem and its variants, including the minimum dominating set problem, the following conjecture is plausible: For any real  $\varepsilon > 0$ , the minimum monopoly problem has no  $(\ln n - \varepsilon)$  approximation unless  $NP \subseteq Dtime(n^{\log \log n})$ . On the other hand, it is known [12] that a greedy algorithm yields a  $(\ln |E| + 1)$  approximation for the minimum monopoly problem. Bermond and Peleg [1] studied some of its modifications, “ $r$ -monopoly” and “self-ignoring” monopoly. Repetitive versions of the local majority polling system were also discussed by several authors [7, 10, 11, 13].

As mentioned above, we can rule the whole system by ruling a small monopoly. By this property, in some applications, a monopoly is a favorite concept, and a (smart) way of monopolizing a given set by modifying the system topology is looked for. Motivated by them, in this paper, we first consider the following problem:

**MONOPOLY VERIFICATION**

**Input:** Two graphs  $G_1 = (V, E_1)$  and  $G_2 = (V, E_2)$  such that  $E_1 \subseteq E_2$ , and a vertex set  $M \subseteq V$ .

**Question:** Does there exist a graph  $G = (V, E)$  such that (1)  $E_1 \subseteq E \subseteq E_2$  and (2)  $M$  is a monopoly in  $G$ ?

In case of YES, we want to compute a graph  $G$  with some additional property. Among such properties, we consider the maximality and minimality.

**MAX-NEIGHBORHOOD MONOPOLY**

**Input:** Two graphs  $G_1 = (V, E_1)$  and  $G_2 = (V, E_2)$  such that  $E_1 \subseteq E_2$ , and a vertex set  $M \subseteq V$ .

**Output:** A graph  $G = (V, E)$  such that (1)  $E_1 \subseteq E \subseteq E_2$ , (2)  $M$  is a monopoly in  $G$ , and (3)  $M$  is not a monopoly in  $G' = (V, E')$  with  $E_1 \subseteq E' \subseteq E_2$  and  $|E'| > |E|$ , if such an  $E'$  exists; NO, otherwise.

**MIN-NEIGHBORHOOD MONOPOLY**

**Input:** Two graphs  $G_1 = (V, E_1)$  and  $G_2 = (V, E_2)$  such that  $E_1 \subseteq E_2$ , and a vertex set  $M \subseteq V$ .

**Output:** A graph  $G = (V, E)$  such that (1)  $E_1 \subseteq E \subseteq E_2$ , (2)  $M$  is a monopoly in  $G$ , and (3)  $M$  is not a monopoly in  $G' = (V, E')$  with  $E_1 \subseteq E' \subseteq E_2$  and  $|E'| < |E|$ , if such an  $E'$  exists; NO, otherwise.

Let us assume that the current system topology is represented by  $G_2 = (V, E_2)$  (resp.,  $G_1 = (V, E_1)$ ). Then the members of  $M$  try to find the minimum-cost links in  $E_2 - E_1$  so that the topology obtained from  $G_1$  by removing (resp., adding) such links secures the adoption of a proposal of  $M$ , if the members of  $M$  pay for the cost of breaking (resp., establishing) links in  $E_2 - E_1$ . This corresponds to the max-neighborhood monopoly (resp., min-neighborhood monopoly) problem.

We note that, if  $E_1 = \emptyset$ , then the max-neighborhood monopoly problem is a maximum subgraph problem (or a minimum edge-deletion problem). On the other hand, if  $G_2$  is complete (i.e.,  $G_2 = K_n$ , where  $n = |V|$ ), the min-neighborhood monopoly problem is a minimum edge-augmentation problem.

Let us then consider the case in which the answer to the monopoly verification problem is NO. In this case, we want to compute a graph  $G$  in which  $M$  controls as many vertices in  $V$  as possible.

**MAX CONTROLLED SET**

**Input:** Two graphs  $G_1 = (V, E_1)$  and  $G_2 = (V, E_2)$  such that  $E_1 \subseteq E_2$ , and a vertex set  $M \subseteq V$ .

**Output:** A graph  $G$  such that (1)  $E_1 \subseteq E \subseteq E_2$  and (2)  $|Cont(G, M)| \geq |Cont(G', M)|$  holds for all  $G' = (V, E')$  with  $E_1 \subseteq E' \subseteq E_2$ .

Section 2 studies the monopoly verification problem. We show that it is polynomially solvable by reducing it to the network flow problem. In Sections 3 and 4 we consider the max- and min-neighborhood monopoly problems, respectively. Although both problems are more general than the monopoly verification problem, we show that they are also polynomially solvable. Finally Section 5 investigates the max controlled set problem. Contrary to the previous problems, it turns out to be intractable, even if either  $G_1$  is empty or  $G_2$  is complete. We finally present a simple approximation algorithm which guarantees the approximation ratio of 2.

For space reasons, proofs of some results are omitted.

## 2 Monopoly Verification Problem

For a graph  $G = (V, E)$  and  $A, B \subseteq V$ , let  $E(A, B) = (A \times B) \cap E = \{(v, w) \in E \mid v \in A, w \in B\}$ . If  $A = \{v\}$  (resp.,  $B = \{w\}$ ), then we simply write  $E(v, B)$  (resp.,  $E(A, w)$ ) instead of  $E(A, B)$ .

Suppose that  $M$  is a monopoly in a graph  $G = (V, E)$ . Let  $U = V \setminus M$  and  $D = E_2 \setminus E_1$ . We consider the following two modifications on  $E$ ; (1) adding edges in  $D(M, M) = (M \times M) \cap D$  to  $E$  and (2) deleting edges in  $D(U, U)$  from  $E$ . Since these modifications do not affect the condition that  $M$  be a monopoly in the graph  $G$ , we can assume in this section that the edge set  $E$  satisfies

$$E \supseteq E_1 \cup D(M, M), \quad (2)$$

$$E \subseteq E_1 \cup D(M, M) \cup D(U, M). \quad (3)$$

For a vertex  $v \in U$ , let

$$\text{deficit}(v) = |N_{G_1}(v) \cap U| - |N_{G_1}(v) \cap M|. \quad (4)$$

By definition,  $v$  is controlled by  $M$  in  $G_1$  if and only if  $\text{deficit}(v) \leq 0$ . Let  $U_>$  and  $U_≤$  be the sets of vertices  $v \in U$  such that  $\text{deficit}(v) > 0$  and  $\text{deficit}(v) \leq 0$ , respectively. Then  $v \in U_≤$  is controlled by  $M$  in any graph  $G$  with  $E$  ( $\supseteq E_1$ ) that satisfies property (B). Thus, we can restrict  $E$  to

$$E(U_≤, M) = E_1(U_≤, M) \quad (\text{i.e., } E \subseteq E_1 \cup D(M, M) \cup D(U_>, M)). \quad (5)$$

Let  $G^+ = (V, E_1 \cup D(M, M))$ . For a vertex  $v \in M$ , let

$$\text{surplus}(v) = |N_{G^+}(v) \cap M| - |N_{G^+}(v) \cap U|. \quad (6)$$

By property (2),  $\text{surplus}(v)$  represents an upper bound on the number of edges  $(v, w) \in D(v, U_>)$  that can be added to  $G_1$ . If  $\text{surplus}(v) < 0$  holds for some  $v$ , we can see that  $v$  is not controlled by  $M$  in any graph  $G = (V, E)$  with  $E_1 \subseteq E \subseteq E_2$ . We thus assume that all vertices  $v \in M$  satisfy  $\text{surplus}(v) \geq 0$ .

We now define a network  $N = (G^* = (V^*, E^*), c : E^* \mapsto \mathbb{R}^+)$  by

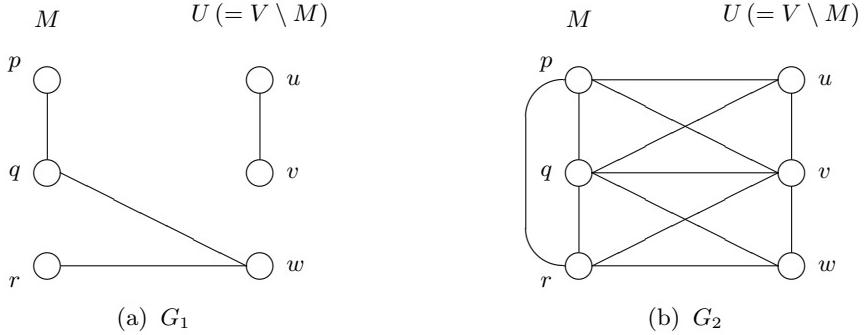
$$V^* = U_> \cup M \cup \{s, t\},$$

$$E^* = E_s \cup E_t \cup D(U_>, M),$$

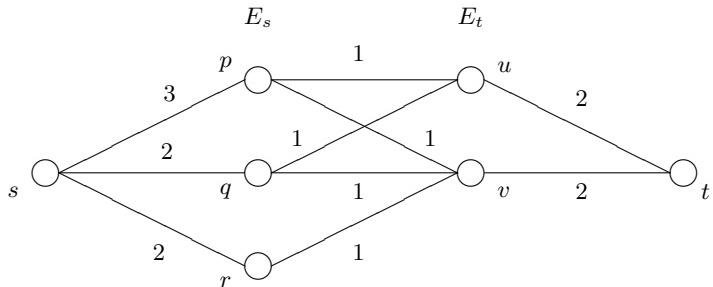
where  $E_s = \{(s, v) \mid v \in M\}$  and  $E_t = \{(w, t) \mid w \in U_>\}$ , and a capacity function

$$c(e) = \begin{cases} \text{surplus}(v) & \text{if } e = (s, v) \in E_s, \\ \text{deficit}(w) & \text{if } e = (w, t) \in E_t, \\ 1 & \text{if } e = (v, w) \in D(U_>, M), \end{cases}$$

For example, let us consider the problem instance given in Figure 1. We can see that the corresponding network  $N$  is represented by Figure 2.



**Fig. 1.** Two graphs  $G_1 = (V, E_1)$  and  $G_2 = (V, E_2)$  with  $E_1 \subseteq E_2$ , where  $V = \{p, q, r, u, v, w\}$  and  $M = \{p, q, r\}$ .



**Fig. 2.** The network  $N = (G^* = (V^*, E^*), c : E^* \mapsto \mathbb{R}^+)$  associated with  $G_1$ ,  $G_2$  and  $M$  in Figure 1.

The following lemma shows our problem can be reduced to a network flow problem in  $G^*$ .

**Lemma 1.** *There exists a graph  $G = (V, E)$  such that (1)  $E_1 \subseteq E \subseteq E_2$  and (2)  $M$  is a monopoly in  $G$ , if and only if the network  $N$  has a maximum  $s$ - $t$  flow whose size is  $\sum_{w \in U_>} \text{deficit}(w)$ .*

*Proof.* Since  $\sum_{w \in V^*} c(w, t) = \sum_{w \in U_>} \text{deficit}(w)$ , no  $s$ - $t$  flow has size greater than  $\sum_{w \in U_>} \text{deficit}(w)$ . Let us assume first that network  $N$  has a maximum  $s$ - $t$  flow whose size is  $\sum_{w \in U_>} \text{deficit}(w)$ . Since  $c(e)$ ,  $e \in E^*$ , is an integer,  $N$  has an integral maximum  $s$ - $t$  flow  $f$ , i.e., for each  $e \in E^*$ ,  $f(e)$  is an integer. Let

$$E = E_1 \cup D(M, M) \cup \{e \in D(U_>, M) \mid f(e) = 1\}.$$

Since  $f$  eliminates all deficits, we can see that  $M$  is a monopoly in the graph  $G = (V, E)$ .

On the other hand, let us assume that  $M$  is a monopoly in the graph  $G = (V, E)$  with  $E_1 \subseteq E \subseteq E_2$ . For each  $w \in U_>$ , we arbitrarily choose  $\text{deficit}(w)$  edges from  $E(w, M) - E_1(w, M)$ , and let  $\Delta(w)$  be the set of  $\text{deficit}(w)$  such edges. Note that  $M$  is a monopoly in  $G' = (V, E')$ , where  $E' = E_1 \cup D(M, M) \cup \bigcup_{w \in U_>} \Delta(w)$ . We now assign nonnegative integers to each  $e \in E^* = E_s \cup E_t \cup D(U_>, M)$  as follows.

$$f(e) = \begin{cases} c^*(e) & \text{if } e = (s, v) \in E_s, \\ \text{deficit}(w) & \text{if } e = (w, t) \in E_t, \\ 1 & \text{if } e = (v, w) \in D(M, U_>) \cap \Delta(w), \\ 0 & \text{if } e = (v, w) \in D(M, U_>) \setminus \Delta(w). \end{cases}$$

where  $c^*(e) = |\{(v, w) \in \Delta(w) \mid w \in U_>\}|$  for  $e = (s, v) \in E_s$ . We can see that this  $f$  is an  $s$ - $t$  flow and its size is  $\sum_{w \in U_>} \text{deficit}(w)$ .  $\square$

For example, Figure 3 shows a maximum flow in the network  $N$  given in Figure 2. This flow corresponds to the graph  $G$  in Figure 3.

Let us note that the size of the network  $N$  satisfies  $|V^*| \leq n+2$ ,  $|E^*| \leq n+m_2$  and  $\max c(e) \leq n$ , where  $n = |V|$  and  $m_2 = |E_2|$ . Since a maximum flow on such a network can be computed in  $\tilde{O}(\min\{(n+m_2)^{3/2}, n^{2/3}(n+m_2)\})$  time [6, 1], we have the following result.

**Theorem 1.** *The monopoly verification problem can be solved in  $\tilde{O}(\min\{(n+m_2)^{3/2}, n^{2/3}(n+m_2)\})$  time.*  $\square$

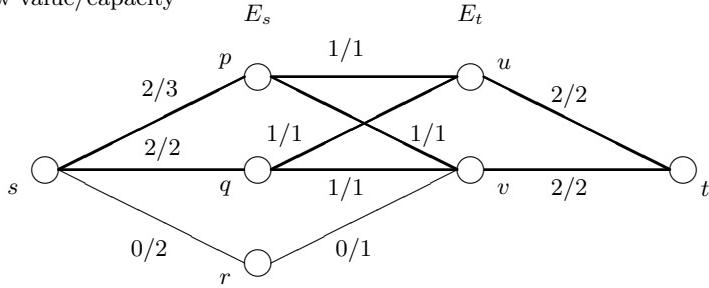
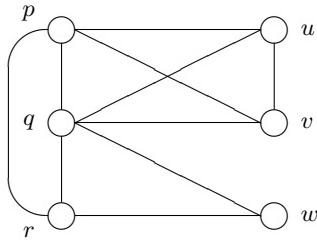
### 3 Max-Neighborhood Monopoly Problem

In this section, we consider the max-neighborhood monopoly problem. This section always assumes that the answer to the monopoly verification problem is “Yes”, i.e., there exists a graph  $G = (V, E)$  such that  $E_1 \subseteq E \subseteq E_2$  and  $M$  is a monopoly in  $G$ . We show that the max-neighborhood monopoly problem can be solved in polynomial time by solving a maximum weighted matching in an associated graph.

Recall the definitions,  $U = V \setminus M$  and  $D = E_2 \setminus E_1$ . Let  $G = (V, E)$  be a solution to the max-neighborhood monopoly problem, where  $E = E_1 \cup \Delta$  with

<sup>1</sup>  $\tilde{O}()$  notation is similar to usual  $O()$  notation except that  $\tilde{O}()$  ignores logarithmic factors.

flow value/capacity

(a)  $N = (G^* = (V^*, E^*), c : E^* \mapsto \mathbf{R}^+)$  $M$  $U (= V \setminus M)$ (b)  $G$ **Fig. 3.** A maximum flow in the network  $N$  given in Figure 2 and the corresponding graph  $G$ .

$\Delta \subseteq D$ . By the maximality of  $\Delta$ ,  $\Delta$  clearly contains  $D(M, M)$ . Let  $U_>$  and  $U_{\leq}$  be as defined in the previous section. For a vertex  $v \in U$  (resp.,  $v \in M$ ), define  $\text{deficit}(v)$  (resp.,  $\text{surplus}(v)$ ) by (4) (resp., (6)). Further, for each  $v \in M$ , we define the “usable surplus,”

$$\text{surplus}^*(v) = \min\{\text{surplus}(v), |D(v, U)|\}. \quad (7)$$

By our assumption,  $\text{surplus}^*(v) \geq 0$  and  $|\Delta(v, U)| = \text{surplus}^*(v)$  holds for each  $v \in M$ . For every maximum  $\Delta$ ,  $|\Delta(M, M \cup U)|$  is of the same size, and hence a maximum  $\Delta$  contains a maximum  $\Delta(U, U)$ .

We now associate a graph  $G^* = (V^*, E^*)$  with  $G_1 = (V, E_1)$ ,  $G_2 = (V, E_2)$  and  $M \subseteq V$ :

$$V^* = V_1 \cup V_2 \cup V_3 \cup V_4,$$

$$E^* = \bigcup_{(v,w) \in D(M \cup U, U)} E_{(v,w)},$$

where

$$V_1 = \{v_1, v_2, \dots, v_{\text{surplus}^*(v)} \mid v \in M\},$$

$$\begin{aligned}
V_2 &= \{v_1, v_2, \dots, v_{|deficit(v)|} \mid v \in U\}, \\
V_3 &= \{x_{e1}, x_{e2}, x_{e3}, x_{e4} \mid e \in D(M, U)\}, \\
V_4 &= \{z_{<v,w>} , z_{<w,v>} \mid (v, w) \in D(U, U)\}, \\
E_{(v,w)} &= \begin{cases} \{(v_i, x_{e1}), (x_{e\ell}, x_{e(\ell+1)}), (x_{e2}, w_j) \mid v_i \in V_1, w_j \in V_2, \ell = 1, 2, 3\} \\ \quad \text{if } e = (v, w) \in D(M, U_>), \\ \{(v_i, x_{e1}), (x_{e\ell}, x_{e(\ell+1)}) \mid v_i \in V_1, \ell = 1, 2, 3\} \\ \quad \text{if } e = (v, w) \in D(M, U_≤), \\ \{(x_{e4}, z_{<v,w>}), (z_{<v,w>} , z_{<w,v>}), (z_{<w,v>} , x_{e'4}) \mid e \in D(M, v), e' \in D(M, w)\} \\ \quad \text{if } (v, w) \in D(U_>, U_>), \\ \{(x_{e4}, z_{<v,w>}), (v_i, z_{<v,w>}), (z_{<v,w>} , z_{<w,v>}), (z_{<w,v>} , x_{e'4}) \mid e \in D(M, v), \\ \quad e' \in D(M, w), v_i \in V_2\} \\ \quad \text{if } (v, w) \in D(U_≤, U_>), \\ \{(x_{e4}, z_{<v,w>}), (v_i, z_{<v,w>}), (z_{<v,w>} , z_{<w,v>}), (z_{<w,v>} , x_{e'4}), (z_{<v,w>} , w_j) \\ \quad \mid e \in D(M, v), e' \in D(M, w), v_i, w_j \in V_2\} \\ \quad \text{if } (v, w) \in D(U_≤, U_≤). \end{cases}
\end{aligned}$$

Here we assume that  $z_{<v,w>} \neq z_{<w,v>}.$  Moreover, let us define a function  $weight : E^* \mapsto \mathbf{R}^+$  by

$$weight(e^*) = \begin{cases} 4L & \text{if } e^* = (x_{e1}, x_{e2}), \\ L & \text{if } e^* = (x_{e3}, x_{e4}), \\ 3L & \text{if } e^* \neq (x_{e1}, x_{e2}), (x_{e3}, x_{e4}) \text{ and } e^* \in E_{(v,w)} : (v, w) \in D(M, U), (8) \\ 3 & \text{if } e^* = (z_{<v,w>} , z_{<w,v>}), \\ 2 & \text{otherwise,} \end{cases}$$

where

$$L > \sum_{e \in D(U, U)} weight(E_e). \quad (9)$$

Here  $weight(T) = \sum_{e^* \in T} weight(e^*)$  for a set  $T \subseteq E^*.$

Note that every  $E_{(v,w)}$  forms a tree, and satisfies  $E_{(v,w)} \cap E_{(v',w')} = \emptyset$  if  $(v, w) \neq (v', w').$  By (8) and (9),

$$weight(e^*) > \sum_{(v,w) \in D(U, U)} weight(E_{(v,w)}) \quad (10)$$

holds for each  $e^* \in E_{(v,w)}$  with  $(v, w) \in D(M, U).$

We now show that a maximum weighted matching  $S$  in  $G^*$  corresponds to a desired graph  $G.$  Let  $Weight$  denote the weight of a maximum weighted matching  $S,$  i.e.,  $Weight = weight(S).$  Let  $\Theta = \Theta_1 + \Theta_2,$  where

$$\Theta_1 = L \sum_{v \in M} surplus^*(v) + L \sum_{v \in U_>} deficit(v) + 5L|D(M, U)| \quad (11)$$

$$\Theta_2 = 3|D(U, U)| + |\Delta(U, U)|. \quad (12)$$

(Recall that  $G = (V, E = E_1 \cup \Delta)$  is a desired graph.)

**Lemma 2.**  $Weight \geq \Theta$  holds.

*Proof.* Let us construct, from  $\Delta$ , a matching  $T$  of  $G^*$  with weight  $weight(T) = \Theta$ . We first introduce mappings  $\alpha$  and  $\beta$  for each edge  $(v, w) \in \Delta(M, U)$ .

Let  $\alpha$  be an arbitrary one-to-one mapping from  $\Delta(M, U)$  to  $V_1$  such that  $\alpha(v, w) = v_i$  for some  $i = 1, 2, \dots, surplus^*(v)$ . Since every  $v \in M$  satisfies  $|\Delta(v, U)| = surplus^*(v)$ ,  $\alpha$  is well-defined. Let  $V' = \{w_1, w_2, \dots, w_{deficit(w)} \mid w \in U_{>}\}$  and  $V'' = \{x_{e3} \mid e \in D(M, U)\}$ . Let  $\beta$  be an arbitrary injective mapping from  $\Delta(M, U)$  to  $V' \cup V''$  such that (i) either  $\beta(v, w) = w_i$  or  $x_{(v,w)3}$  holds for every  $(v, w) \in \Delta(M, U)$ , and (ii) for every  $w_i \in V'$ , there exists an edge  $(v, w) \in \Delta(M, U)$  such that  $\beta(v, w) = w_i$ . Here a mapping  $\chi$  is called *injective* if  $\chi(p) \neq \chi(q)$  holds for  $p \neq q$ . Since  $M$  is a monopoly in  $G$ ,  $\beta$  is also well-defined. These  $\alpha$  and  $\beta$  show how to allocate the surplus on  $M$  to the deficit on  $U$ , where  $\beta(v, w) = x_{(v,w)3}$  means that  $(v, w)$  produces the surplus on  $w$ .

Similarly, for each  $w \in U$ , let  $\gamma_w$  be an injective mapping from  $\Delta(w, U)$  to  $\{w_1, w_2, \dots, w_{-deficit(w)}\} \cup \{x_{(v,w)4} \mid \beta(v, w) = x_{(v,w)3}\}$ , where  $\{w_1, w_2, \dots, w_{-deficit(w)}\} = \emptyset$  if  $deficit(w) \geq 0$ . Intuitively,  $\gamma_w(w, u) = w_i$  means that the surplus on  $w$  is used to add  $(w, u)$  to  $G_1$ . On the other hand,  $\gamma_w(w, u) = x_{(v,w)4}$  means that the surplus on  $v$  which is transferred through the edge  $(v, w)$  to  $w$  is used to add  $(w, u)$  to  $G_1$ .

From these mappings, we define a matching  $T$  in  $G^*$ . For a vertex  $e = (v, w) \in D(M, U)$ , define a set  $T_e$  of edges in  $G^*$  by

$$T_e = \begin{cases} \{(\alpha(e), x_{e1}), (x_{e2}, \beta(e)), (x_{e3}, x_{e4})\} & \text{if } e \in \Delta(M, U) \text{ and } \beta(e) = w_i \text{ for some } i, \\ \{(\alpha(e), x_{e1}), (x_{e2}, \beta(e))\} & \text{if } e \in \Delta(M, U) \text{ and } \beta(e) = x_{e3}, \\ \{(x_{e1}, x_{e2}), (x_{e3}, x_{e4})\} & \text{otherwise,} \end{cases} \quad (13)$$

and for an edge  $e = (w, u) \in D(U, U)$ , define a set  $T_e$  of edges in  $G^*$  by

$$T_e = \begin{cases} \{(\gamma_w(e), z_{<w,u>}), (\gamma_u(e), z_{<u,w>})\} & \text{if } e \in \Delta(U, U), \\ \{(z_{<w,u>}, z_{<u,w>})\} & \text{otherwise.} \end{cases} \quad (14)$$

Let  $T = \bigcup_{e \in D(M \cup U, U)} T_e$ . We can see that  $T$  forms a matching in  $G^*$ . Note that, for an edge  $e = (v, w) \in D(M, U)$ ,  $weight(T_e) = 7L, 6L$  or  $5L$ , which respectively correspond to the first, second, and third cases in (13). Exactly  $\sum_{w \in U_{>}} deficit(w)$  (resp.,  $\sum_{v \in M} surplus^*(v) - \sum_{w \in U_{>}} deficit(w)$  and  $|D(M, U)| - \sum_{v \in M} surplus^*(v)$ ) edges belong to the first case (resp., the second and third cases). Thus,

$$\sum_{e \in D(M, U)} weight(T_e) = \Theta_1. \quad (15)$$

For an edge  $e = (w, u) \in D(U, U)$ ,  $weight(T_e) = 4$  or  $3$ , which respectively correspond to the first and second cases in (14). Exactly  $|\Delta(U, U)|$  (resp.,  $|D(U, U)| - |\Delta(U, U)|$ ) edges belong to the first case (resp., the second case). Thus,

$$\sum_{e \in D(U, U)} weight(T_e) = \Theta_2, \quad (16)$$

which together with (15) implies  $weight(T) = \Theta$ . This completes the proof.  $\square$

To show the opposite inequality, let us show the following lemma.

**Lemma 3.** *Let  $T$  be a matching of  $G^*$  such that  $\text{weight}(T) \geq \Theta_1$ . Then (I5) holds.*

*Proof.* For each edge  $e \in D(M, U)$ ,  $E_e$  forms a tree, and  $T_e$  is a matching in  $E_e$ , where  $T_e = T \cap E_e$ . The weight of a maximum matching in  $E_{(v,w)}$  is  $5L$  when  $\text{surplus}^*(v) = 0$ . When  $\text{surplus}^*(v) > 0$ , the weight is  $7L$  if  $\text{deficit}(w) > 0$ , and  $6L$ , otherwise. However, since  $T$  is a matching, (1) among those with  $\text{surplus}^*(v) > 0$  and  $\text{deficit}(w) > 0$ , at most  $(\sum_{w \in U_>} \text{deficit}(w) T_{(v,w)})$  edges can have weight  $7L$ , and (2) among those with  $\text{surplus}^*(v) > 0$ , at most  $(\sum_{v \in M} \text{surplus}^*(v) T_{(v,w)})$  edges can have weight at least  $6L$ . Since the weight of  $T_e$  is at least  $5L$ , we have

$$\sum_{e \in D(M, U)} \text{weight}(T_e) \leq L \sum_{v \in M} \text{surplus}^*(v) + L \sum_{v \in U_>} \text{deficit}(v) + 5L|D(M, U)|. \quad (17)$$

Moreover, if  $\sum_{e \in D(M, U)} \text{weight}(T_e) < \Theta_1$ , then

$$\sum_{e \in D(M, U)} \text{weight}(T_e) \leq \Theta_1 - L. \quad (18)$$

From (I0), this implies (I5).  $\square$

Let  $T$  be a matching in  $G^*$  such that  $\text{weight}(T) \geq \Theta_1$ . Then the proof of Lemma 3 also shows that  $\bigcup_{e \in D(M, U)} T_e$  gives a desirable  $\Delta'$  in the sense that  $M$  is a monopoly in  $G = (V, E_1 \cup D(M, M) \cup \Delta')$ , where  $\Delta'$  is obtained by reversing the construction of (I3). Moreover, this implies that  $\bigcup_{e \in D(U, U)} T_e$  gives a desirable  $\Delta''$  (i.e.,  $M$  is a monopoly in  $G = (V, E_1 \cup D(M, M) \cup \Delta' \cup \Delta'')$ ), where  $\Delta''$  is obtained by reversing the construction of (I4). More precisely,  $e \in \Delta''$  if and only if  $\text{weight}(T_e) = 4$ . We therefore have the following lemma:

**Lemma 4.** *Weight  $\leq \Theta$  holds.*  $\square$

From Lemmas 2 and 4, we obtain an interesting characterization of the max-neighborhood monopoly problem.

**Corollary 1.** *Weight =  $\Theta$  holds.*  $\square$

Let us note that the size of the graph  $G^*$  satisfies  $|V^*| = O(m_2)$ ,  $|E^*| = O(m_2^2)$  and  $\max \text{weight}(e^*) = O(m_2^2)$ , where  $m_2 = |E_2|$ . Since a maximum weighted matching on such a graph can be computed in  $\tilde{O}(m_2^{5/2})$  time [4], we have the following theorem.

**Theorem 2.** *The max-neighborhood monopoly problem can be solved in  $\tilde{O}(m_2^{5/2})$  time.*  $\square$

## 4 Min-Neighborhood Monopoly Problem

In this section, we consider the min-neighborhood monopoly problem. As in Section 3, we assume in this section that there exists a graph  $G = (V, E)$  such that  $E_1 \subseteq E \subseteq E_2$  and  $M$  is a monopoly in  $G$ .

Recall the definition,  $U = V \setminus M$ . For a vertex  $v \in V$ , let

$$\text{surplus}(v) = |N_{G_1}(v) \cap M| - |N_{G_1}(v) \cap U|, \quad (19)$$

and let  $\text{deficit}(v) = -\text{surplus}(v)$ . Note that these definitions are different from those in the previous sections. By definition,  $v$  is controlled by  $M$  in  $G_1$  if and only if  $\text{surplus}(v) \geq 0$  (i.e.,  $\text{deficit}(v) \leq 0$ ). Let  $M_-$ ,  $M_0$  and  $M_+$  be the sets of vertices  $v \in M$  such that  $\text{surplus}(v) < 0$ ,  $\text{surplus}(v) = 0$  and  $\text{surplus}(v) > 0$ , respectively, and let  $U_-$ ,  $U_0$  and  $U_+$  be the sets of vertices  $v \notin M$  such that  $\text{surplus}(v) < 0$ ,  $\text{surplus}(v) = 0$  and  $\text{surplus}(v) > 0$ , respectively. By definition,  $M = M_- \cup M_0 \cup M_+$  and  $U = U_- \cup U_0 \cup U_+$ .

Let  $G = (V, E)$  be a solution to the min-neighborhood monopoly problem, and let  $E = E_1 \cup \Delta$ , where  $\Delta \subseteq D = E_2 \setminus E_1$ . Since minimizing  $E$  is clearly equivalent to minimizing  $\Delta$ , we discuss properties of  $\Delta$  instead of those of  $E$ . Based on discussions in Section 2, without loss of generality, we can assume

$$\Delta \subseteq D(M, M) \cup D(U_-, M). \quad (20)$$

The following lemma is easy to prove:

**Lemma 5.** *For each  $v \in U_-$ ,*

$$|\Delta(v, M)| = \text{deficit}(v). \quad (21)$$

The following corollary is a direct consequence of the above lemma.

**Corollary 2.**  $|\Delta(U_-, M)| = \sum_{v \in U_-} \text{deficit}(v)$ . □

Corollary 2 implies that, for every minimum  $\Delta$ ,  $|\Delta(U_-, M)|$  is of the same size, and hence  $\Delta$  contains a minimum  $\Delta(M, M)$ .

We now associate a graph  $G^* = (V^*, E^*)$  with  $G_1 = (V, E_1)$ ,  $G_2 = (V, E_2)$  and  $M \subseteq V$  as follows:

$$\begin{aligned} V^* &= V_1 \cup V_2 \cup V_3 \cup V_4 \cup V_5, \\ E^* &= \bigcup_{v \in U_-, w \in M} E_{(v,w)} \cup \bigcup_{v \in M_-} E_v \cup E_a, \end{aligned}$$

where

$$\begin{aligned} V_1 &= \{v_1, v_2, \dots, v_{\text{deficit}(v)} \mid v \in M_-\}, \\ V_2 &= \{v_1, v_2, \dots, v_{\text{deficit}(v)} \mid v \in U_-\}, \\ V_3 &= \{v_1, v_2, \dots, v_{\text{surplus}(v)} \mid v \in M_+\}, \\ V_4 &= \{x_{(v,w)}, y_{(v,w)} \mid (v, w) \in D(U_-, M)\}, \end{aligned}$$

$$\begin{aligned}
V_5 &= \{z_{<v,w>} , z_{<w,v>} \mid (v, w) \in D(M, M)\}, \\
E_{(v,w)} &= \{(v_i, x_{(v,w)}), (x_{(v,w)}, y_{(v,w)}), (y_{(v,w)}, z_{<w,u>}), \\
&\quad (y_{(v,w)}, w_j) \mid v_i \in V_2, w_j \in V_3, z_{<w,u>} \in V_5\} \text{ for } (v, w) \in D(U_-, M_+), \\
E_{(v,w)} &= \{(v_i, x_{(v,w)}), (x_{(v,w)}, y_{(v,w)}), (y_{(v,w)}, z_{<w,u>}), \\
&\quad v_i \in V_2, z_{<w,u>} \in V_5\} \text{ for } (v, w) \in D(U_-, M_0 \cup M_-), \\
E_v &= \{(v_i, z_{<v,w>}) \mid v_i \in V_1, (v, w) \in D(v, M)\} \text{ for } v \in M_-, \\
E_a &= \{(z_{<v,w>} , z_{<w,v>}) \mid z_{<v,w>} , z_{<w,v>} \in V_5\}.
\end{aligned}$$

Here we assume that  $s_{(v,w)} = s_{(w,v)}$  for  $s = x, y$ , and  $z_{<v,w>} \neq z_{<w,v>}$ . We also define a function,  $\text{weight} : E^* \mapsto \mathbb{R}^+$  by

$$\text{weight}(e) = \begin{cases} 1 & \text{if } e \in E_a, \\ 3L & \text{if } e \in E_b, \\ 2L & \text{otherwise,} \end{cases}$$

where  $E_b = \{(x_{(v,w)}, y_{(v,w)}) \mid x_{(v,w)}, y_{(v,w)} \in V_4\}$  and  $L$  is a number greater than  $|E_a|$ .

Let  $S$  be a maximum weighted matching in  $G^*$ , and let  $\text{Weight}$  denote its weight; i.e.,  $\text{Weight} = \text{weight}(S)$ , where  $\text{weight}(T) = \sum_{e \in T} \text{weight}(e)$  for a set  $T \subseteq E^*$ .

Although the proof is skipped due to the space limitation, we can show that computing a maximum weighted matching in  $G^*$  is polynomially equivalent to the min-neighborhood monopoly problem.

**Lemma 6.** *Let  $\text{Weight}$  and  $L$  be as defined above, and let  $\Delta$  be a minimum edge set added to  $G_1$  so that  $M$  is a monopoly in  $G = (V, E_1 \cup \Delta)$ . Then*

$$\text{Weight} - \Theta = |D(M, M)| - |\Delta(M, M)|$$

holds, where

$$\Theta = 3L|D(U_-, M)| + L \sum_{v \in U_-} \text{deficit}(v) + 2L \sum_{v \in M_-} \text{deficit}(v). \quad (22)$$

Let us note that the size of the graph  $G^*$  satisfies  $|V^*| = O(m_2)$ ,  $|E^*| = O(m_2^2)$  and  $\max \text{weight}(e^*) = O(m_2)$ , where  $m_2 = |E_2|$ . Since a maximum weighted matching on such a graph can be computed in  $\tilde{O}(m_2^{5/2})$  time [4], we have the following theorem.

**Theorem 3.** *The min-neighborhood monopoly problem can be solved in  $\tilde{O}(m_2^{5/2})$  time.*  $\square$

## 5 Max Controlled Set Problem

Let us finally consider the max controlled set problem. Unfortunately, this problem is intractable, even if we restrict ourselves to the edge-augmentation and the edge-deletion problems.

**Theorem 4.** *The max controlled set problem is NP-hard, even if  $G_1$  is empty or even if  $G_2$  is a complete graph.*  $\square$

Since the max controlled set problem seems to be intractable, we consider an approximation algorithm. We present a simple approximation algorithm which guarantees an approximation ratio of 2.

For two graphs  $G_1 = (V, E_1)$  and  $G_2 = (V, E_2)$ , and a set  $M \subseteq V$ , we construct two graphs  $G^+ = (V, E \cup E^+)$  and  $G^{++} = (V, E \cup E^{++})$  for  $G_1$  and  $G_2$  by

$$\begin{aligned} E^+ &= E_1 \cup D(M, M), \text{ and} \\ E^{++} &= E_1 \cup D(M, M) \cup D(U, M), \end{aligned}$$

respectively. Here  $U = V \setminus M$ . Let  $W^+$  and  $W^{++}$  be the sets of vertices in  $V$  which are controlled by  $M$  in  $G^+$  and  $G^{++}$ , respectively. The following lemma is immediate from the definitions of  $G^+$  and  $G^{++}$ .

**Lemma 7.** *Let  $W^+$  and  $W^{++}$  be as defined above. Let  $\mathcal{W}$  be a family of sets  $W \subseteq V$  which are controllable by  $M$  in some graph  $G = (V, E)$  with  $E_1 \subseteq E \subseteq E_2$ . Then we have*

$$\begin{aligned} |W^+ \cap M| &= \max_{W \in \mathcal{W}} |W \cap M|, \\ |W^{++} \cap U| &= \max_{W \in \mathcal{W}} |W \cap U|. \end{aligned}$$

**Lemma 8.** *Let  $W^+$  and  $W^{++}$  be as defined above. Let  $W^*$  be the larger of the two, i.e.,  $|W^*| = \max\{|W^+|, |W^{++}|\}$ . Then  $W^*$  satisfies*

$$|W^*| \geq 1/2 \max_{W \in \mathcal{W}} |W|.$$

*Proof.* It follows from Lemma 7 that  $\max_{W \in \mathcal{W}} |W| \leq |W^+| + |W^{++}| \leq 2|W^*|$ .  $\square$

**Theorem 5.** *Given two graphs  $G_1 = (V, E_1)$ ,  $G_2 = (V, E_2)$ , and a set  $M \subseteq V$ , we can compute in polynomial time a graph  $G = (V, E)$  with  $E_1 \subseteq E \subseteq E_2$  such that the size of the set controlled by  $M$  in  $G$  is at least half of that of a maximum controlled set.*  $\square$

## 6 Conclusion

This paper discussed edge augmentation and deletion problems when the number of vertices controlled by a given set  $M$  of vertices is held at maximum. These problems were shown to be NP-complete in general, by a transformation from the maximum independent set problem. However, it can be determined in polynomial time if the addition (or deletion) of a set of edges can make  $M$  control all vertices, by reducing it to a network flow problem.

One can easily extend the positive results in the following way. For a function  $f$  on  $V$ , a vertex  $v \in V$  is called *f-controlled* if  $|N_G(v) \cap M| - |N_G(v) \setminus M| \geq f(v)$ . Then the corresponding problems to monopoly verification, max-neighborhood monopoly and min-neighborhood monopoly problems can be solved in polynomial time by applying the network flow and matching arguments to them, respectively, and the approximation argument also holds for the max  $f$ -controlled set problem. However, the NP-hardness result does not hold for every function  $f$ . For example, if  $f(v) = |V|$  for all  $v \in V$ , then the max  $f$ -controlled set problem is polynomially solvable.

Some problems remain to be addressed in further work. One issue is the search for faster or simpler algorithms for our problems. Another issue is to consider max controlled set problem for special classes of graphs.

*Acknowledgments.* We thank the referees for their helpful suggestions.

## References

1. J-C. Bermond and D. Peleg, The power of small coalitions in graphs, *Proc. SIROCCO'95*, Olympia, Greece, Carleton Univ. Press, 173–184, 1995.
2. D. Fitoussi and M. Tennenholtz, Minimal social laws, *Proc. AAAI'98*, 26–31, 1998.
3. U. Feige, A threshold of  $\ln n$  for approximating set cover, *Proc. STOC'96*, 314–318, 1996.
4. H. N. Gabow and R. E. Tarjan, Faster scaling algorithms for general graph matching problems, *Journal of the ACM*, 38, 815–853, 1991.
5. M. R. Garey and D. S. Johnson, *Computers and Intractability*, Freeman, New York, 1979.
6. A. V. Goldberg and S. Rao, Beyond the flow decomposition barrier, *Journal of the ACM*, 45, 783–797, 1998.
7. Y. Hassin and D. Peleg, Distributed probabilistic polling and applications to proportionate agreement, *Proc. ICALP'99*, LNCS 1644, 402–411, 1999.
8. N. Linial, D. Peleg, Y. Rabinovich and M. Saks, Sphere packing and local majorities in graphs, *Proc. 2nd Israel Symposium on Theoretical Computer Science*, IEEE Computer Soc. Press, 141–149, 1993.
9. C. Lund and M. Yannakakis, On the hardness of approximating minimization problems, *Proc. STOC'93*, 286–293, 1993.
10. T. Nakata, H. Imahayashi, and M. Yamashita, Probabilistic Local Majority Voting for the Agreement Problem on Finite Graphs, *Proc. COCOON*, LNCS 1627, 330–338, 1999.
11. T. Nakata, H. Imahayashi, and M. Yamashita, A Probabilistic Local Majority Polling Game on Weighted Directed Graphs with an Application to the Distributed Agreement Problem, *Networks* (to appear).
12. D. Peleg, Local majority voting, small coalitions and controlling monopolies in graphs: A review, Technical Report CS96-12, Weizmann Institute, 1996.
13. D. Peleg, Size bounds for dynamic monopolies, *Discrete Applied Mathematics*, 86, 263–273, 1998.
14. Y. Shoham and M. Tennenholtz, Emergent conventions in multi-agent systems: initial experimental results and observations, *Proc. KR'92*, 225–231, 1992.
15. Y. Shoham and M. Tennenholtz, On the synthesis of useful social laws for artificial agent societies, *Proc. AAAI'92*, 276–281, 1992.

# Optimal Adaptive Fault Diagnosis of Hypercubes

Andreas Björklund

Department of Computer Science  
Lund University  
[andreas@cs.lth.se](mailto:andreas@cs.lth.se)

**Abstract.** System level fault diagnosis deals with the problem of identifying component failures in a multiprocessor system. Each processor is either *faulty* or *fault-free*, and the objective is to find out the fault status of each processor in the network by letting the processors test each other. A test of a processor by another processor is possible if they are connected in the system. If the tester itself is fault-free, it always reports the fault status of the testee, but if the tester is faulty, the result of the test cannot be trusted.

We show that for the hypercube multiprocessor system of dimension  $n$ , in which at most  $n$  processors are faulty, adaptive diagnosis is possible using at most  $2^n + n - 1$  tests, which improves earlier bounds and is optimal. We also present an algorithm which diagnoses the hypercube in 4 testing rounds, where each processor is scheduled for at most one test of each round.

## 1 Introduction

In the fault diagnosis model of Preparata, Metze, and Chien [10], each processor is either *faulty* or *fault-free*. The problem is to locate all faulty processors in the system by letting the processors test each other. A test of a processor  $v$  (the *testee*) by another processor  $u$  (the *tester*) denoted  $(u, v)$  is possible iff  $u$  and  $v$  are connected in the system. The outcome of a test is 0(1) if the tester diagnosed the testee as fault-free(faulty). If the tester itself is fault-free it always reports the fault status of the testee, but if the tester is faulty, the result of the test cannot be trusted. In adaptive diagnosis, you may choose what tests to be made based on the outcomes of previous tests. Furthermore, it is assumed that the fault-status of a processor does not change during the diagnosis.

We show that for the hypercube multiprocessor system of dimension  $n$ , in which at most  $n$  processors are faulty, adaptive diagnosis is possible using at most  $2^n + n - 1$  tests, which is optimal. [8] shows the best previous bound of  $2^n + \frac{3n}{2}$  tests. We also present an algorithm which diagnoses the hypercube in 4 testing rounds, where each processor is scheduled for at most one test of each round. This result improves on the 11 round algorithm in [8].

## 2 Historical Notes

The problem of fault diagnosis of multiprocessor systems was originated by Preparata, Metze and Chien [10]. They show that sometimes it is not possible to diagnose a multiprocessor system even though all possible tests are made. Two necessary conditions for a system to be diagnosable is pointed out. The first one is that a strict majority of the processors must be fault-free, and the second one says that there may not be more faulty processors than the minimum degree of the system. The degree of a processor is the number of processors connected to it, and the minimum degree of a system is the minimum of the degrees of all its processors.

The early research focused on non-adaptive diagnosis, i.e. you must decide exactly which tests you will make prior to the diagnosis. Nakajima [9] is the first to propose adaptive diagnosis, in which you may choose your tests depending on the results of previous tests. This method increases the efficiency of fault diagnosis drastically. [10] shows that a complete system on  $n$  processors, i.e. all pairs of processors are connected, where at most  $k$  processors are faulty, requires  $kn$  tests in the non-adaptive case. In contrast, Blecher [4] shows that  $n + k - 1$  tests are necessary and sufficient for the adaptive diagnosis of the same system. Note that this lower bound applies to all systems, since by removing connections you will not add any testing possibilities. The question of whether this bound is tight for some sparse systems as well is partially answered to the affirmative in [7], in which some simple systems with at most a constant number of faulty processors are considered.

Various kinds of fault diagnosis of the hypercube system are studied in [1], [3], and [5]. The first results on adaptive diagnosis of the hypercube is obtained by Feng et al. [6]. They present an algorithm using at most  $2^n(\lfloor \log n \rfloor + 2)$  tests, embedded in at most  $n + 4$  testing rounds. Kranakis and Pelc [8] shows that  $2^n + \frac{3n}{2}$  tests are sufficient, and that adaptive diagnosis can be carried out in a constant number of rounds. We improve on their results by showing that  $2^n + n - 1$  tests are always enough, and 4 testing rounds suffice. Our methods differ from the ones in [8].

## 3 Preliminary Definitions

In the following the hypercube system of processors will be modelled by an undirected graph in which the nodes represent processors, and the edges represent connections between pairs of processors. Let  $P$  be the graph consisting of two connected nodes, then the *hypercube of dimension n*, denoted  $H_n$ , is defined as the cartesian product  $H_n = P \times H_{n-1}$ , where  $H_1 = P$ . From this definition, it is clear that the hypercube  $H_n$  has  $2^n$  nodes and that it can be divided into two identical hypercubes isomorphic to  $H_{n-1}$ , let us call them  $A$  and  $B$ , s.t. every node in  $A$  is connected to exactly one node in  $B$ , and vice versa. We say that  $(A, B)$  is a *mirror decomposition* of  $H_n$  for any such partitioning, and define the corresponding mirror function  $A2B$  mapping every node of  $A$  to its neighbour in

$B$ . Analogously,  $B2A$  is defined as the inverse function. For any set  $S$  of nodes, we let  $\rho(S)$  be the number of faulty nodes contained in  $S$ . We know the *correctness* of a set  $K$  of nodes, if for all nodes in  $K$  we know whether they are faulty or not.

## 4 Adaptive Fault Diagnosis in $2^n + n - 1$ Tests

The lower bound of  $2^n + n - 1$  tests in [4] for the complete graph on  $2^n$  nodes with at most  $n$  faulty ones, applies to the hypercube  $H_n$  as well. We show that this bound is tight for the hypercube.

### 4.1 Some Definitions and Lemmas

An *honest trail*  $T = (v_1, v_2, \dots, v_m)$  in a graph is a multiple of distinct connected nodes along with a collection of test results,  $(v_i, v_{i+1}) = 0$  for all  $i = 1, 2, \dots, m-1$ .

**Lemma 1.** *Let  $G = (V, E)$  be a graph with  $\rho(V) \leq k$ , and  $T = (v_1, v_2, \dots, v_m)$  be an honest trail in  $G$ . If  $\rho(V \setminus T) = w$ , then all  $v_i$  are fault-free for  $i > k - w$ .*

*Proof.* The proof is by contradiction. Assume  $v_j$  faulty, then  $v_{j-1}$  is faulty as well, since the latter node has diagnosed  $v_j$  as fault-free. Repeating the argument, we conclude that all  $v_l$  for  $l \leq j$  are faulty, but we know that there is at most a total of  $k$  faulty nodes in the graph, so  $j + w \leq k$ . Thus,  $v_i$  cannot be faulty for  $i > k - w$ .  $\square$

**Corollary 1.** *The last node  $v_{k+1}$  of an honest trail  $T = (v_1, v_2, \dots, v_{k+1})$  in a graph with at most  $k$  faulty nodes, is fault-free.*

A *vertex cut*  $C$  of a connected graph  $G = (V, E)$  is a subset of  $V$  s.t. the graph induced by removing  $C$  from  $G$  is no longer connected. The size of the smallest vertex cut is called the *connectivity* of the graph. It is a well known fact that the hypercube of dimension  $n \geq 2$  has connectivity  $n$ .

**Lemma 2.** *Let  $G = (V, E)$  be a graph with connectivity  $k$  and  $\rho(V) \leq k$ . Suppose we know the correctness of a subset  $K$  of the nodes  $V$ , and at least one of them is fault-free, and we also know an honest trail  $T = (v_1, v_2, \dots, v_m)$ , s.t.  $T \cap K = \emptyset$ , then*

- In at most  $|V| - |K| - |T| + \min(|T|, \rho(V \setminus K) + 1, k - \rho(K))$  tests, we can diagnose all of  $G$ .
- If in addition, we know a node  $w \notin K \cup T$  s.t.  $(v_m, w) = 1$ , we can diagnose all of  $G$  in at most  $|V| - |K| - |T| + \min(|T|, \rho(V \setminus K), k - \rho(K) - 1)$  tests.

*Proof.* If  $|T| > k - \rho(K)$ , then by lemma 1 we know that the last nodes on the trail must be fault-free. Remove these from the trail and put them into  $K$  (since they are correctly diagnosed). Thereby we have assured that  $|T| + \rho(K) \leq k$ . As long as  $K \neq V$ , we wish to extend the set  $K$  of diagnosed nodes. It can be

accomplished by repeating the following procedure until the correctness of all nodes are learned. If  $\rho(K) < k$ , we know that there must be a fault-free node  $u$  in  $K$  adjacent to some node  $v$  in  $(V \setminus (K \cup T)) \cup \{v_1\}$ . This is because the set consisting of all nodes in  $T$  except  $v_1$  and the  $\rho(K)$  faulty nodes in  $K$ , cannot be a vertex cut simply because its size is less than the connectivity of  $G$ . Make the test  $(u, v)$  to diagnose  $v$  and put it into  $K$ . We distinguish between two cases. If  $v = v_1$  and the outcome of the test was 1, we remove  $v_1$  from the trail  $T$  and rename the remaining nodes of the trail so that the node formerly known as  $v_{i+1}$  now is called  $v_i$  for  $1 \leq i \leq m - 1$ . If  $v = v_1$  was found fault-free though, all nodes in  $T$  are removed and put into  $K$ , since they all must be fault-free, leaving an empty trail. In the second case, when  $v \neq v_1$ , and if the result of the test was 1, and  $|T| + \rho(K) = k$ , the last node of the trail must be fault-free according to lemma 1, and therefore it too is removed from  $T$  and put into  $K$ . If we ever get  $\rho(K) = k$ , we immediately know that all nodes in  $V \setminus K$  must be fault-free. Either way, eventually  $K = V$ , and we are done.

The tests made are easily counted, since every node not initially in  $K$  is tested at most once, and the nodes on the original trail  $T$  are tested in order, i.e.  $v_i$  is tested before  $v_j$  for  $i < j$ , and we stop testing the nodes on the trail either when we find a fault-free one, or all turned out to be faulty. If only  $l < m$  of the nodes on the original trail  $T$  are faulty, the last nodes  $v_i$  for  $i > l + 1$  are not tested at all since they must be fault-free if  $v_{l+1}$  is. This also proves the small improvement in the number of tests in the second statement of the lemma, since in this case, when only  $l < m$  nodes on the trail are faulty,  $w$  must be faulty. On the other hand, if  $w$  was diagnosed fault-free, then all  $v_i \in T$  must be faulty.  $\square$

## 4.2 The Main Result

Our proof that a hypercube graph of dimension  $n \geq 3$  with at most  $n$  faulty nodes can be adaptively diagnosed in at most  $2^n + n - 1$  tests, is by induction. In fact, we show that something slightly stronger holds. First, we show the result for the hypercube of dimension 2.

**Lemma 3.** *The hypercube of dimension 2 with at most 2 faulty nodes, either cannot be diagnosed at all, and this can be established in 4 tests, or it is diagnosed in 4 tests when no nodes are faulty, or at most 5 tests when some nodes are faulty.*

*Proof.* The proof is by extensive case analysis. The hypercube of dimension 2 is a ring of four nodes. Call the nodes on the ring  $r_i; 0 \leq i < 4$ , where  $r_i$  is connected to  $r_{i+1 \text{ mod } 4}$ . Make all four tests along one direction of the ring, i.e.  $(r_i, r_{i+1 \text{ mod } 4}); 0 \leq i < 4$ . If all four tests resulted in 1, the ring cannot be diagnosed, since either  $r_0$  and  $r_2$  are faulty ( $r_1$  and  $r_3$  are fault-free) or  $r_1$  and  $r_3$  are faulty ( $r_0$  and  $r_2$  are fault-free). Still, note that it is sufficient to establish the correctness of just any one of the four nodes in order to diagnose all of them. If all tests on the ring resulted in 0 though, all four nodes must be fault-free and we are done. If exactly one test resulted in 1, assume w.l.o.g.  $(r_0, r_1) = 1$  and note that  $r_0$  and  $r_3$  are fault-free and  $r_1$  is faulty, so just use  $r_3$  to diagnose  $r_2$ .

and we are done. If exactly two tests resulted in 1, there are two cases. Either two tests next to each other on the ring, or two node disjoint tests resulted in 1. Assume in the first case w.l.o.g. that  $(r_0, r_1) = (r_1, r_2) = 1$  and note that  $r_0$  and  $r_3$  are fault-free and  $r_1$  is faulty, so just use  $r_3$  to diagnose  $r_2$  and we are done. In the second case, assume w.l.o.g.  $(r_0, r_1) = (r_2, r_3) = 1$  and note that  $r_0$  and  $r_2$  are fault-free, whereas  $r_1$  and  $r_3$  are faulty. Finally, if exactly three tests resulted in 1, assume w.l.o.g.  $(r_0, r_1) = 0$  and note that  $r_1$  is fault-free and  $r_2$  is faulty. Make the test  $(r_1, r_0)$ . If the result was 0,  $r_0$  is fault-free and  $r_3$  faulty, otherwise  $r_3$  is fault-free and  $r_0$  is faulty. Either way, we are done.  $\square$

The previous lemma provides the fundamental brick in our inductive proof.

**Theorem 1.** *The hypercube  $H_n$  of dimension  $n \geq 3$  with at most  $n$  faulty nodes can be adaptively diagnosed in at most  $2^n + n - 1$  tests if precisely  $n$  nodes are faulty, and in  $2^n + l$  tests if  $l < n$  nodes are faulty.*

*Proof.* Let  $(A, B)$  be a mirror decomposition of  $H_n$ . Since the hypercube is hamiltonian, it is possible to find a node disjoint path  $p = (v_1, v_2, \dots, v_{n+1})$  in  $A$  when  $n \geq 3$ . Make the tests  $(v_i, v_{i+1})$  along the path for  $i = 1, 2, \dots$  until either a test resulted in 1, or all tests resulted in 0. In the first case, there is an  $m \leq n$  s.t.  $(v_m, v_{m+1}) = 1$  and  $(v_i, v_{i+1}) = 0$  for  $1 \leq i < m$ . Thus, there must be at least one faulty node in  $A$  (one of  $v_m$  and  $v_{m+1}$ ), and therefore, there are at most  $n - 1$  faulty nodes in  $B$ . Diagnose  $B$ , using the induction thesis, or lemma 3 in the event of  $n = 3$ . If  $B$  could not be diagnosed in the latter case, we immediately know at least two fault-free nodes in  $A$  since  $B$  contains two faulty nodes according to the proof of lemma 3. Use one of these to diagnose a node in  $B$  and thereby gain complete knowledge of the correctness of all four nodes in  $B$ . Finally, let a fault-free node in  $A$  test one of  $v_m$  and  $v_{m+1}$  in order to find out which of them is faulty. At most nine tests are made this way and three faults are located, which is consistent with the theorem. If  $B$  can be diagnosed though, the induction thesis for  $n > 3$ , and lemma 3 for  $n = 3$ , assures that at most  $2^{n-1} + \rho(B)$  tests were made diagnosing  $B$ . If precisely  $n - 1$  faulty nodes were found in  $B$ , we once again only need to find out which one of the nodes  $v_m$  and  $v_{m+1}$  is faulty, which clearly can be made in less than  $2^n + n - 1$  tests for  $n \geq 3$ . If fewer than  $n - 1$  faults were found though, apply lemma 2 with  $T = (v_1, v_2, \dots, v_m)$  and  $K = B$ , and note that the assumptions of the second statement of lemma 2 are fulfilled since  $(v_m, v_{m+1}) = 1$ . Thus the number of tests made is at most  $2^n + \rho(V)$  when  $\rho(V) < n$ , and  $2^n + n - 1$ , when  $\rho(V) = n$  as claimed. One case remains to be proven, namely when all the initial  $n$  tests along  $p$  returned 0. But according to corollary 1,  $v_{n+1}$  in this case must be fault-free. Apply lemma 2 with  $T = (v_1, v_2, \dots, v_n)$  and  $K = \{v_{n+1}\}$ . The first statement of the lemma assures that at most  $2^n + \rho(V)$  tests are made when  $\rho(V) < n$ , and  $2^n + n - 1$  tests when  $\rho(V) = n$ .  $\square$

## 5 Adaptive Fault Diagnosis in 4 Rounds

The efficiency of adaptive fault diagnosis can of course be measured counting the number of tests needed as in the previous section. For practical purposes,

this is a poor measurement though. A more adequate one, due to the parallel implementation possibilities, is the number of testing rounds needed to diagnose the graph. In each round, every node is allowed to participate in at most one test, either as a tester or a testee. In [2], it was shown that the complete graph with a majority of fault-free nodes, can be diagnosed in 10 rounds. [8] showed that for the hypercube of dimension  $n$ , with at most  $n$  faulty nodes, 11 rounds suffice. The best lower bound, to our knowledge, is that 2 rounds is not enough. The number of tests made after 2 rounds is at most  $2^n$ , which violates the lower bound of  $2^n + n - 1$  tests for  $n > 1$ . However, we present a diagnosing scheme in 4 rounds.

### 5.1 Some Definitions and a Lemma

Two arcs  $a = (u_1, u_2)$  and  $b = (u_3, u_4)$  are said to *overlap* if  $u_i = u_j$  for some  $1 \leq i \leq 2, 3 \leq j \leq 4$ . A *3-round testing scheme* of a graph  $G = (V, E)$  is a tuple  $(T, R)$  where  $T = (V, A)$  is a directed subgraph of  $G$ , and  $R : A \rightarrow \{1..3\}$  is an arc colouring function s.t. for all distinct pairs of overlapping arcs  $a, b \in A$ ,  $R(a) \neq R(b)$ . Define the *i-th round* of  $(T, R)$  as  $A_i = \{a | a \in A, R(a) = i\}$ . Let  $H = (V_1, E_1)$  be a subgraph of  $G$ , then  $T|H$  is the subgraph of  $T$  on  $V_1$  containing those arcs  $(u, v) \in A$  s.t.  $(u, v) \in E_1$ . We say that a 3-round testing scheme  $(T_n, R)$  for  $H_n$  is *recursively hamiltonian* for  $n \geq 2$  if

- $T_n$  contains a hamiltonian cycle  $C \subseteq A$  for  $H_n$ .
- There are at least two distinct arcs  $a_1, a_2 \in A_2 \cap C$  s.t. no arc in  $A_3$  overlap  $a_1$  or  $a_2$ .
- If  $n > 2$ , there is a mirror decomposition  $(A, B)$  of  $H_n$  s.t.  $(T_n|A, R)$  and  $(T_n|B, R)$  are recursively hamiltonian.

The first and third property ensure the existences of hamiltonian cycles which we will use in our algorithm. The second property is merely added to show the existence of a 3-round testing scheme having the other two.

**Lemma 4.** *There is a recursively hamiltonian 3-round testing scheme  $(T_n, R)$  for all hypercubes  $H_n$  for  $n \geq 2$ .*

*Proof.* For  $H_2$ , we let  $T_2$  consist of a directed cycle of arcs  $a_1, a_2, a_3, a_4$  on its 4 nodes. We define  $R(a_1) = R(a_3) = 1$ , and  $R(a_2) = R(a_4) = 2$ . It is easy to verify that the three properties of the definition for recursively hamiltonian holds for this construction. We proceed by induction on  $n$ . Assume there is a recursively hamiltonian 3-round testing scheme  $S_{n-1} = (T_{n-1}, R)$  for  $H_{n-1}$ , and construct one  $S_n = (T_n, R)$  for  $H_n$  as follows. Let  $(A, B)$  be a mirror decomposition of  $H_n$ . Use the description of  $S_{n-1}$  to build recursively hamiltonian 3-round schemes  $S_A = (T_A, R)$  for  $A$ , and  $S_B = (T_B, R)$  for  $B$ , s.t. if  $(u_1, u_2)$  is an arc in  $T_A$ ,  $(A2B(u_2), A2B(u_1))$  is an arc in  $T_B$ . By the mirror symmetry, it is clear that if  $(v_1, v_2)$  is an arc in  $T_B$ ,  $(B2A(v_2), B2A(v_1))$  is an arc in  $T_A$ . Furthermore, choose the colouring function  $R$  so that  $R((u_1, u_2)) = R((A2B(u_2), A2B(u_1)))$ , for all arcs  $(u_1, u_2)$  in  $T_A$ . In words, this means that we embed the recursively

hamiltonian 3-round scheme in  $A$  and  $B$  in opposite directions, but otherwise identically. The second property of the definition for recursively hamiltonian states that there are two arcs  $a_1, a_2$  in  $T_A$  on a hamiltonian cycle of  $A$ , whose endpoints do not belong to any arcs  $a$  with  $R(a) = 3$ . Our embeddings  $T_A$  and  $T_B$  ensure that for  $a_1 = (u_1, u_2)$ ,  $b_1 = (A2B(u_2), A2B(u_1))$  belongs to  $T_B$ , with  $R(a_1) = R(b_1)$ , and  $b_1$  is an arc having the second property in the definition of recursively hamiltonian. Similarly, we can construct  $b_2$  from  $a_2$ . Thus none of the nodes  $u_1, u_2, A2B(u_2)$ , or  $A2B(u_1)$  are part of an arc  $a$  with  $R(a) = 3$ , so we may add the arcs  $x_1 = (u_1, A2B(u_1))$  and  $x_2 = (A2B(u_2), u_2)$ , and define  $R(x_1) = R(x_2) = 3$ , to conclude that  $T_n = T_A + T_B + x_1 + x_2$  is hamiltonian. The arcs  $a_2$  and  $b_2$  obey the second property of recursively hamiltonian, and the third one follows from our inductive construction.  $\square$

## 5.2 The Main Result

Our algorithm to diagnose the hypercube in 4 rounds is partially static, since the first three rounds are always the same. The tests of the fourth round though, may be completely different depending on the outcomes of the tests scheduled earlier.

**Theorem 2.** *The hypercube  $H_n$  for  $n \geq 3$  can be adaptively diagnosed in 4 testing rounds.*

*Proof.* Construct a recursively hamiltonian 3-round testing scheme  $S_n = (T_n, R)$  for  $H_n$  from lemma 4. Divide the arcs  $A^{T_n}$  of  $T_n$  into its  $i$ th round components  $A_i^{T_n} = \{a | a \in A^{T_n}, R(a) = i\}$  and make the tests corresponding to arcs in  $A_i^{T_n}$  in round  $i$ . To put it more formally, if  $(u_1, u_2) \in A_i^{T_n}$ , then the test  $(u_1, u_2)$  is made in the  $i$ th testing round. By the definition of a 3-round testing scheme, no two tests in the same round use the same node. By the third property of the definition of recursively hamiltonian, we know there is a mirror decomposition  $(A, B)$  s.t. the tests made in  $A$  ( $B$ ) form a recursively hamiltonian 3-round testing scheme for  $A$  ( $B$ ). Thus the first property of the definition for recursively hamiltonian ensures that among the tests made, there are hamiltonian cycles of tests  $C_A$  in  $A$ , and  $C_B$  in  $B$ . If all tests along  $C_A$  ( $C_B$ ) resulted in 0, we know from corollary 1 that all nodes along the cycle, i.e. all of  $A$  ( $B$ ), are fault-free. In this event, assume w.l.o.g that  $C_A$  was fault-free, then we can schedule the tests  $(a, A2B(a))$  for all nodes  $a$  in  $A$  in the fourth round. Since all nodes in  $A$  are fault-free, we know the correctness of both  $A$  and  $B$  after the fourth round. On the other hand, if there were tests  $t_A$  along  $C_A$ , and  $t_B$  along  $C_B$  both resulting in 1, we know there must be faulty nodes in both  $A$  and  $B$ . Hence there are at most  $n - 1$  faulty nodes in each of  $A$  and  $B$ , and we can use recursion on  $A$  and  $B$  to find out which tests are to be carried out in round 4. The only problem left is the bottom of the recursion when  $A$  and  $B$  are of dimension 2. A case analysis very similar to the one in the proof of lemma 2, which is omitted, show us how to overcome this obstacle.  $\square$

## 6 Conclusions

We have shown that locating faulty processors in a multiprocessor system of  $2^n$  processors, of which at most  $n$  are faulty, can be done as efficiently in the hypercube system as in the complete system. Still, removal of any connection between some pair of processors from the hypercube system leaves the system undiagnosable. By undiagnosable we mean that it may be that even if you make all possible tests in the system, there is some processor whose fault status you cannot decide. This is because the two processors at the end of the removed connection have lower degree than the number of possible faulty processors, which violates the second necessary condition for diagnosability in [10], mentioned in the historical notes section. In this sense, the hypercube structure is optimal for the adaptive diagnosis problem.

We also showed that it is possible to schedule tests in just 4 testing rounds, to adaptively diagnose the hypercube. For the complete system on  $2^n$  processors of which at most  $n$  are faulty, this can be strengthened to 3 rounds. Simply let the two first rounds constitute of a lot of cycles of length greater than  $n$ . Many of these will be found fault-free and can be used in the third round to diagnose the other cycles. It is still open whether 3 testing rounds is sufficient for the hypercube system. It should be noted that the number of tests in our 4 round testing scheme may be as many as  $2^{n+1}$ , whereas the 11 round construction in [8] uses at most  $2^n + (n+1)^2$  tests.

## References

1. J. Armstrong and F. Gray, Fault diagnosis in a Boolean n-Cube array of microprocessors, *IEEE Transactions on Computers* 30 (1981), 587-590.
2. R. Beigel, W. Hurwood and N. Kahele, Fault diagnosis in a flash, Proc. 36th Symp. on Found. of Comp. Sci. (1995), 571-580.
3. P. Berman and A. Pelc, Distributed probabilistic fault diagnosis for multiprocessor systems, Dig. 20th Int. Symp. Fault-Tolerant Computing, IEEE Computer Society Press (1990), 340-346.
4. P. M. Blecher, On a logical problem, *Disc. Math.* 43 (1983), 107-110.
5. D.M. Blough, G.F. Sullivan and G.M. Masson, Efficient diagnosis of multiprocessor systems under probabilistic models, *IEEE Transactions on Computers* 41 (1992), 1126-1136.
6. C. Feng, L.N. Bhuyan and F. Lombardi, Adaptive system-level diagnosis for hypercube multiprocessors, *IEEE Transactions on Computers* 45 (1996), 1157-1170.
7. E. Kranakis, A. Pelc and A. Spatharis, Optimal adaptive fault diagnosis for simple multiprocessor systems, *Networks* 34 (1999), 206-214.
8. E. Kranakis and A. Pelc, Better Adaptive Diagnosis of Hypercubes, submitted.
9. K. Nakajima, A new approach to system diagnosis, Proc. 19th Allerton Conf. Commun. Contr. and Computing (1981), 697-706.
10. F. Preparata, G. Metze and R. Chien, On the connection assignment problem of diagnosable systems, *IEEE Transactions on Electron. Computers* 16 (1967), 848-854.

# Fibonacci Correction Networks<sup>\*</sup>

Grzegorz Stachowiak

Institute of Computer Science, University of Wrocław,  
Przesmyckiego 20, 51-151 Wrocław, Poland  
[gst@ii.uni.wroc.pl](mailto:gst@ii.uni.wroc.pl)

**Abstract.** In this paper we construct sorting comparator networks which correct a fixed number  $t$  of faults in a sorted sequence of length  $N$ . We study two kinds of such networks. One construction yields a fault tolerant unit that attached at the end of any comparator sorting network makes the whole network a sorting one resistant to  $t$  passive faults. The second network can be used to ‘repair’ a sorted sequence in which at most  $t$  entries were changed (no fault tolerance is required). The new results of this paper are constructions of comparator networks of depth  $1.44 \cdot \log N$  for these problems which is less than the depths of networks described by previous authors [3], [4], [5]. The construction of the networks is practical for small  $t$ . The numbers of comparators used by our networks are shown to be reducible to values optimal up to a constant factor.

## 1 Introduction

Sorting is one of the most fundamental problems of computer science. A classical approach to sort a sequence of keys is to apply a comparator network. Apart from a long tradition, comparator networks are particularly interesting due to potential hardware implementations. They can be also implemented as sorting algorithms for parallel computers.

In our approach sorted elements are stored in registers  $r_1, r_2, \dots, r_N$ . Registers can be indexed with integers or elements of other linearly ordered sets. In this paper a convenient convention is indexing registers with sequences of integers  $\mathbf{x} = (x_1, x_2, \dots, x_k)$  ordered lexicographically. A set of all registers having the same first coordinate  $x_1$  is called *row* labeled  $x_1$ . A set of all registers having the same all but first coordinates we call *column* labeled with the sequence of fixed coordinates. The first coordinate of a register we call its *level* in the column. We define operation  $\circ$  on sequences of integers

$$(x_1, \dots, x_k) \circ (y_1, \dots, y_l) = (x_1, \dots, x_k, y_1, \dots, y_l).$$

By  $|\mathbf{x}|$  we denote the length of  $\mathbf{x}$ .

A *comparator*  $[i : j]$  is a simple device connecting registers  $r_i$  and  $r_j$  ( $i < j$ ). It compares the numbers they contain and if the number in  $r_i$  is bigger, it swaps

---

\* Partially supported by KBN grant 8 T11C 032 15 and by University of Wrocław grant 2320/W/IIn/99.

them. The general problem is the following. At the beginning of the computations the input sequence of keys is placed in the registers. Our task is to sort the sequence of keys according to the linear order of register indexes applying a sequence of comparators. The sequence of comparators is determined before the computations. We assume that comparators connecting disjoint pairs of registers can work in parallel. Thus we arrange this sequence of comparators into a series of comparator *layers* which are sets of comparators connecting disjoint pairs of registers. The total time needed by a comparator network to perform its computations is proportional to the number of layers of the network called its *depth*.

Much research concerning sorting networks have been done in the past. Their main goals were to minimize the depth and the total number of comparators. The most famous results are asymptotically optimal AKS [1] sorting network of depth  $O(\log N)$  and more ‘practical’ Batcher [2] network of depth  $\sim \frac{1}{2} \log^2 N$  (all logarithms in this paper are of base 2). Another well known result we are going to apply in this paper is Yao’s [6] construction of an almost optimal network to select  $t$  smallest (or largest) entries of a given input of size  $N$  (*t-selection* problem). His network has depth  $\log N + (1 + o(1)) \log t \log \log N$  and  $\sim N \log t$  comparators which matches lower bounds for that problem ( $t \ll N$ ).

In this paper we deal with two problems concerning comparator networks. One of them is to construct a comparator network which is a unit correcting  $t$  passive faults (see [7]) in any sorting network (some comparators are faulty and do nothing). Such a unit can be attached to any sorting network e.g. AKS (as a number of its last layers) so that the whole network is a sorting one resistant to  $t$  faults. The unit has to correct all the faults present in the sorting network and be resistant to all errors present in itself. Such a correcting unit we call *t-fault-tolerant* network. The best result concerning such networks is that of Piotrów [4], who constructed asymptotically optimal network of depth  $O(\log N + t)$  having  $O(Nt)$  comparators. The exact constants hidden behind these big  $O$ -h’s were not determined, but since Piotrów uses network [5] the constant in front of  $\log N$  in  $O(\log N + t)$  is at least 2.

The other problem is to sort an almost sorted sequence. Let us consider for example a large sorted database with  $N$  entries. In some period of time we change  $t$  entries and want to have it sorted back. We design a specialized comparator network of a small depth to ‘repair’ the ordering and avoid using costly general sorting networks. Such a network to sort back a sorted sequence in which at most  $t$  changes were made we call *t-correction* network. The best known general result here is network of Kik, Kutylowski, Piotrów [3] of depth  $4 \log N + O(\log^2 t \log \log N)$ .

The networks in [3] [4] are based on a nice construction by Schimmler and Starke [5] of a 1-correction network of depth  $2 \log N$  having  $3.5N$  comparators. Our goal is to reduce the constant in front of  $\log N$  in the depth, which is most essential if  $t$  is small and  $N$  big. We present *t-fault-tolerant* and *t-correction*

networks that for fixed  $t$  have depths  $\sim \alpha \log N = \log_{\frac{1+\sqrt{5}}{2}} N$ , where

$$\alpha = \frac{1}{\log_2 \frac{1+\sqrt{5}}{2}} = 1.44\dots$$

This way our networks have smaller depths than any correction networks described by previous authors.

For sorting networks the following useful lemma called Zero–One Principle holds:

**Lemma 1 ((zero–one principle)).** *A comparator network is a sorting network if and only if it can sort any input consisting only of 0's and 1's.*

This lemma is the reason, why from now on we only consider inputs consisting only of 0's and 1's. Below we formulate analogous lemmas for fault tolerant and correction networks.

We say 0 (1) is *disturbed* if it is changed to 1 (0). Resulting 1 (0) we call *displaced*. A sequence of 0's and 1's produced from a sorted sequence by disturbing  $t$  or less entries we call *t-disturbed*.

**Lemma 2.** *A comparator network is a t-fault-tolerant network if and only if for any  $x \leq t$  it can sort any  $x$ -disturbed input if we remove any set of  $t - x$  comparators.*

**Lemma 3.** *A comparator network is a t-correction network if and only if it can sort any t-disturbed input.*

We define *dirty area* for 0-1 sequences contained in the registers during computations of a comparator network. Dirty area is the minimal set of subsequent registers such that below these registers (in registers with lower indexes) there are only 0's and above there are only 1's. A  $t$ -disturbed input in which only 0's are disturbed we call *t-partially-disturbed*. A comparator network that can reduce dirty area size to at most  $\Delta$  for any  $x$ -partially-disturbed input having  $t - x$  faulty comparators we call *(t, Δ)-partial-fault-tolerant*. Comparator network that can reduce dirty area size to at most  $\Delta$  for any  $t$ -partially-disturbed input we call *(t, Δ)-partial-correction*. For both networks the output is  $t$ -partially-disturbed, because a 1 can only increase the index of its register during computations. The final size of dirty area  $\Delta = \Delta(N, t)$  is some function of  $N$  and  $t$ .

## 2 One Disturbed Position

In this section we consider sorting of 1-disturbed inputs. For simplicity we assume the input is 1-partially-disturbed, i.e. has a single disturbed 0. So we have one displaced 1 at this position. We describe a comparator network  $F_N$  correcting any 1-partially-disturbed input of size  $N$ . Due to symmetry of the network the case of displaced 0 follows directly from this case.

First we recall the definition of Fibonacci numbers  $f_k$ :

$$f_0 = f_1 = 1,$$

$$f_k = f_{k-2} + f_{k-1}.$$

We define the numbers  $\varphi_k, \psi_k$  and  $\vartheta_k$  behaving similarly to  $f_k$ :

$$\varphi_0 = \varphi_1 = \psi_0 = \psi_1 = 1,$$

$$\psi_k = \varphi_{k-2} + \psi_{k-1},$$

$\varphi_k$  = the largest odd number smaller or equal  $\psi_k$ .

$$\vartheta_k = 2\psi_k - \varphi_k$$

Let  $\text{LG}(n)$  be the smallest  $p$  such that  $\varphi_p \geq n$ . Let  $r_1, r_2, \dots, r_N$  be registers. The network  $F_N$  consists of  $d = \text{LG}(N)$  subsequent layers  $L_1, L_2, \dots, L_d$  such that:

$$L_p = \{[2i + p : 2i + p + \varphi_{d-p}] | i \in \mathcal{Z}\}$$

The way we define  $L_p$  requires a few words of comment. From all comparators in the definition of  $L_p$  only those exist whose end registers are well defined and belong to the set of all registers. This convention is maintained for the rest of the paper. Now we prove that the comparator network defined above is a 1-correction network indeed and estimate how many layers it has asymptotically.

**Fact 1**  $d = \text{LG}(N) \sim \log_{(1+\sqrt{5})/2} N = \alpha \log N$

*Proof.* The Fact follows directly from inequalities  $f_{k-1} \leq \varphi_k \leq f_k$  which can be easily proven.

To see how the network works introduce first some definitions. The highest register containing 0 of the input we call *border*. The *distance* between displaced 1 and the border is the difference between indexes of border and register containing displaced 1. Our network proves to reduce this distance very efficiently. It ends computations, when the distance is guaranteed to be 0. The fact the network  $F_N$  is really a 1-correction network follows directly from the following lemma applied to layer  $d$ .

**Lemma 4.** *After applying the first  $l$  layers of  $F_N$  the distance between single displaced 1 and the border is smaller than  $\psi_{d-l+1}$ . If this 1 is in a register  $r_i$  for which  $i = l \bmod 2$ , then this distance is smaller than  $\varphi_{d-l}$ .*

*Proof.* We proceed by induction on  $l$ . For  $l = 0$  the lemma is obvious, because  $\varphi_d \geq N$ . Assume that the lemma holds for  $l - 1$  and prove it for  $l$ . Let  $i$  be the index of the register containing displaced 1 just before we apply layer  $l$ . If the displaced 1 is not moved by layer  $l$ , then two cases are possible. The first is  $i \neq l \bmod 2$  and from inductive hypothesis for  $l$  the distance is smaller than

$\varphi_{d-l+1}$  which is not bigger than  $\psi_{d-l+1}$ . In the second case  $i = l \bmod 2$  the fact 1 is not moved by layer  $l$  means that the distance is smaller than  $\varphi_{d-l}$ . If this 1 is moved then its distance is reduced by  $\varphi_{d-l}$  from some value smaller than  $\psi_{d-l+2}$ . Thus after applying layer  $l$  this distance is smaller than

$$\psi_{d-l+2} - \varphi_{d-l} = \psi_{d-l+1}.$$

As we see from the last lemma at any moment of computations of network  $F_N$  there is a set of registers below the border such that a single displaced 1 contained one of these registers is guaranteed to get to the border at the end of the computations. After the first  $l$  layers this set contains all registers  $r_i$  below the border (in registers with lower indexes) and in the distance from the border smaller than

- $\psi_{d-l+1}$  if  $i \neq l \bmod 2$
- $\varphi_{d-l}$  if  $i = l \bmod 2$

We call this set the *correction area*. In later considerations we set the border register somewhere below the highest 0. It is easy to see that in such case a single displaced 1 in correction area also is guaranteed to get to the border or to some higher register.

Obviously last lemma implies the following corollary since at the end of computations of  $F_N$  the distance is zero (and due to symmetry of the network):

**Corollary 1.** *Comparator network  $F_N$  is 1-correction network of depth  $\alpha \log N$ .*

### 3 Partial Fault Tolerant Network

In this section we define a  $(t, t^2 + t)$ -partial-fault-tolerant network  $T(s, N, t)$  of a small depth. The network is constructed for a parameter  $s$  being an arbitrary integer constant. Later in this paper we show how having this network we can easily produce a  $t$ -fault-tolerant network of a similar depth.

The main idea of construction of  $T(\cdot)$  is to apply a number of networks  $F_{N'}$ . As we proved  $F_{N'}$  guarantees sorting any input with a single displaced 1. If the number of displaced 1's is bigger they can disturb each other to get to the border. At least one of them gets to the border but others do not have to. We solve this problem moving displaced 1's that drop out of correction area to another  $F_{N'}$  network slightly delayed in comparison to the previous one. This way 1's that lost their chance to get to the border in one  $F_{N'}$  regain it in another  $F_{N'}$ .

Now we describe the whole network in a more formal manner. Register indexes are ordered pairs  $(i, j)$  for  $i \in \{1, \dots, N/t\}, j \in \{1, \dots, t\}$ . It is easy to see that if we change all displaced 1's to 0's then in each column we have the highest 0 almost at the same level in its column (the levels can differ by 1). Network  $T(s, N, t)$  consists of two parts. The first part is preprocessing consisting of the sequence of layers:

$$P_1, P_2, \dots, P_{3t},$$

where

$$P_q = \{(i, 2j + q) : (i, 2j + q + 1)] | i, j \in \mathcal{Z}\}.$$

**Fact 2** After applying the first part of  $T(\cdot)$  to  $x$ -partially-disturbed input all displaced 1's are pushed to registers with biggest possible coordinate  $j$  (if the number of faults in is not bigger than  $t - x$ ). In other words:  $r(i, j)$  contains displaced 1 implies  $r(i, j + 1)$  also contains a displaced 1.

Let  $d = \text{LG}(N/t)$ . The second part of the network does the main work and is the sequence of layers:

$$L_1, L_2, \dots, L_s, C_s, L_{s+1}, \dots, L_{2s}, C_{2s}, L_{2s+1}, \dots, L_{3s}, C_{3s}, L_{3s+1}, \dots$$

where

$$L_p = \{(2i + p, j) : (2i + p + \varphi_{d-p+2s(t-j)}, j)] | i, j \in \mathcal{Z}\}$$

and

$$C_q = \{(2i + q + 1, j) : (2i + q + \vartheta_{d-q+2s(t-j)+1}, j - 1)] | i, j \in \mathcal{Z}\}$$

From now on  $\vartheta_k = 1$  for  $k < 0$ . The second part of network  $T(s, N, t)$  has altogether  $(1 + 1/s)\text{LG}(N) + 2(s + 1)(t - 1)$  layers. As we see the layers  $L_p$  are layers of  $F_{N/t}$  inside columns, and layers  $C_q$  roughly speaking move displaced 1's to the next delayed column when they are beyond correction areas in their columns.

Now we analyze what happens to an  $x$ -partially-disturbed input in the network  $T(\cdot)$ . We assign to each 1 during computations the property of being or not being *active*. Just after the first part we switch each displaced 1 to be active and each not displaced 1 not to be active. We define parameter  $b$ . At the beginning  $b$  is the index of the level of register containing the highest 0 in column  $t$  if we change all displaced 1's to 0's. An active 1 stops being active at the moment a comparator moves it to level  $i \geq b$ . At the same moment  $b$  is decreased by one. To each displaced 1 we assign an integer value  $v$ . First we define *destination column index*  $u$  of displaced 1 in a given moment of computations. To do it we change all other active 1's to 0's, repair all faulty comparators in the network and fix  $b$  as it is at this moment of computations. Then we continue computations of the network. If the 1 gets to level  $i \geq b$ , then destination column index  $u$  is the index of the column from which comparator moves it to this level. If displaced 1 does not get to such level at all, then the index  $u$  is set to be 0. The current value  $v$  of an active 1 is equal to the minimum of all values  $u$  assigned to this 1 till the considered moment of computations. When the 1 stops to be active, its value remains unchanged till the end of computations. It is not hard to see from the definition, that at the beginning of the second part each 1 has value equal to the index of its column.

The following facts describe behavior of values assigned to 1's:

**Fact 3** If an active 1 is stopped by a passive fault, then its value decreases by 1 or does not change.

*Proof.* We prove, that its destination column index does not change or decreases by 1. We trace that stopped 1 for  $2s+1$  layers after stop repairing all the faults it can encounter and changing all other active 1's to 0's. Even if destination column index  $u$  of this 1 before it is stopped is equal to the label of its column, after it is stopped the index is smaller. A single displaced 1 that has such smaller  $u$ , after layer  $L_q$  is in register  $r_{2i+q+1}$  and treated by layer  $C_q$  moves to the next column (having label smaller by 1). It is easy to see that during these computations this displaced 1 goes to the next column into a register on higher level, than at the moment it was stopped. The next column because of its delay, at the moment considered 1 gets to it, is at the same or earlier phase of its computations, as column in which the stop occurred was at the moment of stop. It proves that the index  $u$  of 1 does not decrease by more than one.

**Fact 4** *Assume an active 1 is stopped by another active 1 and its value decreases. In such a case its value becomes to be not smaller than the value of 1 causing the delay decreased by 1.*

*Proof.* There is no difference for displaced 1 between being stopped by a passive fault and another displaced 1. Just before one active 1 stops another they must have the same destination column index.

**Lemma 5.** *Network  $T(s, N, t)$  reduces the dirty area of any  $x$ -partially-disturbed input ( $x \leq t$ ) to at most  $t^2 + t$  registers if it has at most  $t - x$  passive faults.*

*Proof.* Putting together the facts one can see that at the end of computations 1's that were displaced at the beginning of the second part of the network have values  $v_1, v_2, \dots, v_x$ . Without loss of generality we can assume that they form a not increasing sequence. Because of the Facts the difference  $v_i - v_{i+1}$  is not bigger than the number of faults 1 with  $v_{i+1}$  encountered increased by one. Since  $v_1 = t$ , we have that  $v_x \geq 1$  and 1 having value  $v_x$  is not active at the end of the second part. It gives dirty area of size not bigger than  $t^2 + t$  since  $b$  is decreased  $x$  times during the computations.

## 4 Partial Correction Network

Now we define a  $(t, ct(\log N)^{c_s \log t})$ -partial-correction network  $C(s, N, t)$ , where  $s$  is an integer constant and  $c_s$  depends on  $s$ . This network has depth  $\alpha(1 + 1/s) \log N + c_s(1 + o(1)) \log t \log \log N$ . We show later in this paper how from this network we can obtain a  $t$ -correction network of almost the same depth. For this section we change denotation  $\psi_i$  to  $\psi(i)$  (the same for  $\varphi$  and  $\vartheta$ ).

Before we begin to construct the network  $C(\cdot)$  we prove a lemma about network  $F_N$  on which the construction is based. As we know network  $F_N$  successfully corrects one displaced 1. The lemma describes its behavior if the number of displaced 1's is bigger.

**Lemma 6.** Assume that in a given moment of computations not less than  $t$  displaced 1's are in the correction area of  $F_N$ . In such a case after the next layer of  $F_N$  at least  $t/2$  displaced 1's are in the correction area. Consequently after  $s$  layers at least  $t/2^s$  displaced 1's remain in the correction area.

*Proof.* The reason displaced 1 can drop off the correction area is that a comparison between this 1 and another displaced 1 is made. In such case the other 1 remains in correction area.

The main idea of construction  $C(\cdot)$  is to have a number of disjoint networks  $F_{N'}$ . At the beginning all displaced 1's are moved to a few networks  $F_{N'}$  (other become free from displaced 1's). Each  $s$  steps displaced 1's that drop out from correction area in one  $F_{N'}$  are moved to another  $F_{N'}$  not containing previously any displaced 1's. These moved 1's are in correction area of their new network  $F_{N'}$ , because the new  $F_{N'}$  is delayed by  $s + 1$  steps. In the delayed  $F_{N'}$  there is at most fraction  $1 - 1/2^s$  of displaced 1's from the previous  $F_{N'}$ . Thus the total delay cannot grow very much because in the subsequent networks  $F_{N'}$  maximal numbers of displaced 1's go down exponentially. In fact this idea is similar to that applied in [3]. The changes consist in applying  $F_N$  network and putting  $C_q$  layers not every second step, but less frequently. The following simple combinatorial fact says us, that in the our construction the number of networks  $F_{N'}$  is small.

**Fact 5** The number of nondecreasing sequences  $j_1, j_2, \dots, j_k$  for  $0 \leq k \leq K$  and  $1 \leq j_l \leq J$  is equal:

$$\binom{J+K}{K} = O(J^K)$$

*Proof.* The number is the same as the number of nondecreasing sequences of integers  $0 \leq j_l \leq J$  of length  $K$  which is the same as the number of increasing sequences of integers  $1 \leq j_l \leq J + K$ .

Now we define network  $C(\cdot)$  in a more formal way. Let  $K = -\log_{1-1/2^s} t$ ,  $J = \lceil \text{LG}(N) \rceil + K$ . In the network  $C(s, N, t)$  indexes of registers have the form  $(n') \circ \mathbf{j} \circ (J + \tau)$ . In this denotation  $\tau \in \{1, \dots, t\}$ ,  $\mathbf{j} = (j_1, \dots, j_k)$  is a nondecreasing sequence of integers  $j_l \in \{1, \dots, J\}$  of length at most  $K$  and  $n' \in \{1, \dots, N'\}$ , where  $N' = \frac{N}{(K+J)t}$ . As in the case of  $T(\cdot)$ , we can change all displaced 1's into 0's. There are at most two (differing by 1) levels of the highest 0's in a column. We treat the level just below these levels as the border between 0's and 1's for the needs of this algorithm. We exclude from the considerations all displaced 1's which are moved to registers above the border level.

The network  $C(s, N, t)$  consists of two parts. The first part uses a selector for the  $t$  largest entries [6] to each row of registers. After the first part, displaced 1's in all rows below the border get to registers  $R(n', \tau + J)$ . Indexes of these registers are lexicographically biggest in each row. This first part has depth  $\sim c_s \log t \log \log N$ . The constant  $c_s$  grows as  $-\frac{1}{\ln(1-1/2^s)} \sim 2^s$ .

Let  $d = \text{LG}(N')$ . The second part consists of the sequence of layers

$$L_1, L_2, \dots, L_s, C_s, L_s, L_{s+1}, \dots, L_{2s}, C_{2s}, L_{2s+1}, \dots, L_{3s}, C_{3s}, L_{3s+1}, \dots$$

where

$$\begin{aligned} L_p &= \{(2i+p) \circ \mathbf{j} \circ (\tau + J) : (2i+p + \varphi(d + (s+1)|\mathbf{j}| - p)) \circ \mathbf{j} \circ (\tau + J)\}, \\ C_q &= \{(2i+q+1) \circ \mathbf{j} \circ (\tau + J) : \\ &\quad (2i+q+1 + \vartheta(d + (s+1)|\mathbf{j}| - q+1)) \circ \mathbf{j} \circ \left(\frac{q}{s} - |\mathbf{j}|, \tau + J\right)\} \\ &\cup \{(2i+q) \circ \mathbf{j} \circ (\tau + J) : \\ &\quad (2i+q + \varphi(d + (s+1)|\mathbf{j}| - q)) \circ \mathbf{j} \circ \left(\frac{q}{s} - |\mathbf{j}|, \tau + J\right)\}. \end{aligned}$$

Altogether we have  $(1 + 1/s)(\text{LG}(N') + K) + (s+1)K$  layers in the second part.

In this network again layers  $L_p$  represent layers of  $F_{N'}$  inside the columns (similarly to the  $T(\cdot)$ ). Layers  $C_q$  represent transfers of disturbed 1's beyond correction area to columns not containing displaced 1's. From the way layers  $C_q$  are defined we see that only one transfer to a given column can occur during the whole time of computations. Displaced 1's are transferred from column  $\mathbf{j} \circ (\tau + J)$  to column  $\mathbf{j}' \circ (\tau + J)$  and  $|\mathbf{j}'| = |\mathbf{j}| + 1$  (since  $\mathbf{j}' = \mathbf{j} \circ (q/s - |\mathbf{j}|)$ ). All transferred 1's after the transfer are on a level in the distance not bigger than  $\varphi(d + (s+1)|\mathbf{j}'| - q - 1)$  from the border level. Thus they are in correction area for their new column. At most fraction  $1 - 1/2^s$  of displaced 1's is transferred. Because of this in general we have the following fact:

**Fact 6** *A column with the index  $\mathbf{j} \circ (\tau + J)$  contains not more than  $t \cdot (1 - 1/2^s)^{|\mathbf{j}|}$  displaced 1's.*

The fact above is the reason, we do not need columns for  $|\mathbf{j}| > K$ . Even if they were present, no displaced 1's would get to them. Because all displaced 1's are at the end of computations on or above border level also the following fact holds:

**Fact 7** *The second part of the network reduces the dirty area to at most three rows.*

This way the network  $C(\cdot)$  reduces dirty area to at most  $3t \binom{J+K}{K} = ct(\log N)^{c_s \log t}$  registers. This proves the following lemma:

**Lemma 7.** *Comparator network  $C(s, N, t)$  is a  $(t, ct(\log N)^{c_s \log t})$ -partial-correction network for some constant  $c_s$  depending on  $s$ . This network has depth*

$$\alpha \left(1 + \frac{1}{s}\right) \log N + c_s(1 + o(1)) \log t \log \log N$$

## 5 Fault Tolerant and Correction Networks

Now we show how having partial-fault-tolerant and partial-correction networks we can obtain fault-tolerant and correction networks of almost the same depth.

The solutions presented in this section are intended to be as simple as possible and the author believes the reader can find solutions with a bit better constants.

The problem that is often encountered in construction of comparator networks is sorting inputs with dirty areas of small size. Assume we can reduce dirty area of  $t$ -disturbed sequence of 0's and 1's to size  $\Delta$ . The question is how many layers and comparators a comparator network needs to 'clean' this dirty area. We have two versions of this question. One if we require fault-tolerance the other if we do not. This question is answered by easy to prove lemmas:

**Lemma 8.** *Assume that there exists a  $t$ -fault-tolerant network  $X_N$  that for input size  $N$  has depth  $\delta(N, t)$  and  $\gamma(N, t)$  comparators. Then there exists a comparator network that sorts any  $x$ -disturbed input with a dirty area of size at most  $\Delta$  if it has not more than  $t - x$  faulty comparators. This network has depth  $2\delta(2\Delta, t)$  and  $\frac{N}{\Delta}\gamma(2\Delta, t)$  comparators.*

**Lemma 9.** *Assume that there exists a  $t$ -correction network  $X_N$ , that for input of size  $N$  has depth  $\delta(N, t)$  and  $\gamma(N, t)$  comparators. Then there exists a comparator network that sorts any  $t$ -disturbed input with a dirty area of size at most  $\Delta$ . This network has depth  $2\delta(2\Delta, t)$  and  $\frac{N}{\Delta}\gamma(2\Delta, t)$  comparators.*

*Proof of both lemmas.* We index the registers with integers  $1, \dots, N$ . The network consists of two parts  $\delta(2\Delta, t)$  layers each. The first part consists of networks  $X_{2\Delta}$  on each set of registers:

$$S_{2i} = \{r_{2i\Delta+1}, r_{2i\Delta+2}, \dots, r_{2i\Delta+2\Delta}\}.$$

The second part is are the networks  $X_{2\Delta}$  on each set of registers:

$$S_{2i+1} = \{r_{(2i+1)\Delta+1}, r_{(2i+1)\Delta+2}, \dots, r_{(2i+1)\Delta+2\Delta}\}.$$

This network cleans the dirty area because this area is contained in at least one  $S_i$ .

Now having these cleaning networks we formulate the main result of this section. We are going to prove is that to produce a good  $t$ -fault-tolerant(-correction) network it is enough to construct  $(t, \Delta)$ -partial-fault-tolerant(-correction) network  $Y_N$  having small depth and reasonably small function  $\Delta$  and  $t$ -fault-tolerant(-correction) network  $X_N$  of not too big depth and small number of comparators. In such case we can construct  $t$ -fault-tolerant(-correction) network of almost the same depth as  $Y_N$  and having roughly speaking twice as many comparators as  $X_N$  has. We call these reductions Refinement Lemmas. We formulate and prove them at once for fault-tolerant and correction networks.

**Lemma 10 ((refinement lemma)).** *Assume we have a comparator network  $Y_N$  which is  $(t, \Delta)$ -partial-fault-tolerant(-correction) network of depth  $\delta'(N, t)$  ( $\Delta = \Delta(N, t)$ ). We have also a  $t$ -fault-tolerant(-correction) network  $X_N$  of depth  $\delta(N, t)$  and having  $\gamma(N, t)$  comparators. Then for any  $M$  there exists a  $t$ -fault-tolerant(-correction) network for any input size  $N$  of depth  $\Delta = \Delta(Nt/M, t)$*

$$\delta(M, t) + \delta' \left( \frac{Nt}{M}, t \right) + 2\delta \left( \frac{4M\Delta}{t} + 2M, t \right)$$

with the number of comparators not bigger than:

$$\frac{N}{M}\gamma(M, t) + \frac{Nt}{M}\delta'\left(\frac{Nt}{M}, t\right) + \frac{Nt}{2M\Delta+Mt}\gamma\left(\frac{4M\Delta}{t} + 2M, t\right).$$

*Proof.* Let indexes of registers be pairs  $(i, j)$  ( $i \in \{1, \dots, N/M\}$ ,  $j \in \{1, \dots, M\}$ ). Our network consists of three main parts.

In the first part we apply  $X_M$  in each row separately. This requires  $\delta(M, t)$  layers and  $\frac{N}{M}\gamma(M, t)$  comparators. The result of this part is that displaced 0's are moved to the first  $t$  columns, and displaced 1's are moved to last  $t$  columns (except maybe one row).

In the second part we use two copies of  $Y_{Nt/M}$ . The first copy is reversed upside-down to deal with displaced 0's and is applied to all registers of the first  $t$  columns. The second copy is applied to all registers of last  $t$  columns to deal with the displaced 1's. This requires  $\delta'\left(\frac{Nt}{M}, t\right)$  layers and at most  $\frac{Nt}{M}\delta'\left(\frac{Nt}{M}, t\right)$  comparators. The result of this part is that the dirty area is reduced to at most  $2\frac{M\Delta}{t} + M$  registers.

The third part is cleaning network (based on  $X_N$ ) for dirty area  $\frac{2M\Delta}{t} + M$  which requires  $2\delta\left(\frac{4M\Delta}{t} + 2M, t\right)$  layers and  $\frac{Nt}{2M\Delta+Mt}\gamma\left(\frac{4M\Delta}{t} + 2M, t\right)$  comparators.

Now we show how we can use refinement lemmas to construct fault-tolerant and correction networks of a small depth. In the construction of  $t$ -fault-tolerant network we apply the Piotrów's network [5].

**Theorem 8.** *There exists a constant  $c$  such that for an arbitrary  $s$  there exists a  $t$ -fault-tolerant network of depth:*

$$\alpha\left(1 + \frac{1}{s}\right)\log N + c\log\log N + (2s + c)t$$

having  $O(Nt)$  comparators.

*Proof.* We defined  $(t, t^2 + t)$ -partial-fault-tolerant network  $T(s, N, t)$ , which has depth  $\alpha(1 + 1/s)\log N + (2s + c')t$ . Theorem follows from Refinement Lemma applied to  $X_N$  being Piotrów's network,  $Y_N = T(N, s, t)$ ,  $M = t\log N$ .

The above network is practical for small  $t$ . If we fix  $t$  and take  $s = \sqrt{\log N}$ , then we get a  $t$ -fault-tolerant network of depth  $\alpha\log N + O(\sqrt{\log N})$ .

Similarly as for fault tolerant networks we can now construct a  $t$ -correction network applying Refinement Lemma.

**Theorem 9.** *For any integer  $s$  there exists a  $t$ -correction network of depth*

$$\alpha\left(1 + \frac{1}{s}\right)\log N + c'_s(\log t\log\log N)^2.$$

for some constant  $c'_s$  depending on  $s$ .

*Proof.* We apply Refinement Lemma taking  $Y_N = C(\cdot)$ ,  $X_N$ -Batcher network,  $M = t \log N$ .

This network has depth  $\alpha(1+1/s) \log N + o(\log N)$  if  $t = o\left(2\sqrt{\log N / \log \log N}\right)$ . We can take  $s = \log \log \log N$  and since  $c'_s = \Omega(c_s^2)$  we obtain the following corollary:

**Corollary 2.** *For any  $t$  there exists a  $t$ -correction network of depth*

$$\alpha \left(1 + \frac{1}{\log \log \log N}\right) \log N + c \log^2 t \log \log^4 N \sim \alpha \log N.$$

We can also apply Refinement Lemma once again taking the network from previous theorem for  $s = 1$  as  $X_N$ . We put  $Y_N = C(s, N, t)$ ,  $M = t \log N$  and get the following corollary.

**Corollary 3.** *For any integers  $s, t$  there exists a  $t$ -correction network of depth*

$$\alpha \left(1 + \frac{1}{s}\right) \log N + c''_s \log t \log \log N + o(\log \log N).$$

for some constant  $c''_s$  depending on  $s$ .

Unfortunately it is not clear if this corollary improves the bound on  $t$  for which we can make a correction network of depth  $\sim \alpha(1 + 1/s) \log N$ , because the construction works well only for  $t \ll N$ .

## 6 Minimizing Number of Comparators

First we should know what the minimal numbers of comparators for  $t$ -fault-tolerant and  $t$ -correction networks are. Any  $t$ -fault tolerant network has at least  $t$  comparators going from any register different from the highest one to registers with higher indexes (to make it impossible to have them all faulty for 1-disturbed input). So it has at least  $(N - 1)t = \Omega(Nt)$  comparators. Any  $t$ -correction network has to be a  $t$ -selector which forces it to have  $\Omega(N \log t)$  comparators [6]. These asymptotic lower bounds on the numbers of comparators in correction networks prove to be achieved.

A  $t$ -fault-tolerant having asymptotically optimal number of comparators is  $t$ -fault-tolerant network from the previous section. It has depth  $\alpha(1 + 1/s) \log N + O(\log \log N + st)$ .

An optimal  $t$ -correction network we construct using the Refinement Lemma. Similar techniques to those we use in this section can be applied to reduce numbers of comparators of practical correction networks but not to make this paper too long we do not describe how to do it. The simplest way to make these practical constructions is to use Batcher network instead of AKS in what follows. Unfortunately we were not able to find a  $t$ -correction network with asymptotically optimal number of comparators without using AKS network, so

our further constructions are not practical. First we construct a network that is asymptotically optimal in the sense of the number of comparators but is not in the sense of depth.

**Lemma 11.** *There exists a  $t$ -correction network that for some constant  $c$  and any input size  $N$  has depth  $cN \log t/t$  and at most  $cN \log t$  comparators.*

*Proof.* Let AKS denote a sorting network which has depth  $\frac{c}{2} \log t$  for input of size  $2t$  [1]. It has at most  $\frac{c}{2}t \log t$  comparators. We index registers with integers  $1, \dots, N$ . We define sets of registers:

$$S_i = \{r_{(i-1)t+1}, r_{(i-1)t+2}, \dots, r_{(i-1)t+2t}\}.$$

Our network consists of  $2N/t - 1$  parts  $c \log t$  layers each. Each part consists of AKS networks on register sets  $S_i$ . Thus we apply AKS subsequently to

$$S_1, S_2, \dots, S_{N/t}, S_{(N/t)-1}, \dots, S_1.$$

It is easy to see that what we constructed is really a  $t$ -correction network.

When we have the  $t$ -correction network from the last lemma we can put it as  $X_N$  to Refinement Lemma taking  $M = \frac{t \log N}{\log t}$ . As  $Y_N$  we can use AKS which is a  $(t, 0)$ -partial-correction network. This way we obtain the following corollary:

**Corollary 4.** *There exists a  $t$ -correction network of depth  $O(\log N)$  having  $O(N \log t)$  comparators.*

Further on we can take correction network from the last lemma as  $X_N$ ,  $C(\cdot)$  as  $Y_N$  and  $M = t \log N$ . As a result by Refinement Lemma we obtain the following corollary:

**Corollary 5.** *For any integer  $s$  there exists a  $t$ -correction network of depth*

$$\alpha(1 + 1/s) \log N + c'_s \log t \log \log N$$

*for some constant  $c'_s$  depending on  $s$  which has  $O(N \log t)$  comparators.*

## 7 Conclusions

We constructed  $t$ -fault-tolerant and  $t$ -correction networks of depths  $\sim \alpha \log N$  for fixed  $t$ . This is less than depth of 1-correction network found by Schimmler and Starke [5]. Network  $T(\cdot)$  seems to be better for practical purposes although it is worse than  $C(\cdot)$  for combinations of  $N$  and  $t$  where  $N$  is big and  $t \geq \log N$ . Some considerations we did not include in this paper seem to indicate that the following conjecture is true. This conjecture was originally posed by Mirek Kutyłowski – authors only contribution is the constant  $\alpha$ .

*Conjecture 1.* The lower bound for depth of 1-correction network is

$$\alpha \log N - c.$$

for some small constant  $c$ .

Because the author was unable to find 2-correction networks of depth asymptotically better than  $T(\cdot)$ , he dares to pose another conjecture concerning 2-correction networks.

*Conjecture 2.* The lower bound for depth of 2-correction network is

$$\alpha \log N + c\sqrt{\log N}$$

for some constant  $c > 0$ .

## Acknowledgments

Author wishes to thank Mirek Kutyłowski, Krzysiek Loryś and Marek Piotrów for presenting the problems, helpful discussions and their encouragement to write this paper. Author also thanks Mirek Kutyłowski for many valuable remarks that improved the presentation.

## References

1. M. Ajtai, J. Komolós, E. Szemerédi, Sorting in  $c \log n$  parallel steps, *Combinatorica* 3 (1983), 1-19.
2. K.E. Batcher, Sorting networks and their applications, in *AFIPS Conf. Proc.* 32 (1968), 307-314.
3. M. Kik, M. Kutyłowski, M. Piotrów, Correction Networks, in *Proc. of 1999 ICPP*, 40-47.
4. M. Piotrów, Depth Optimal Sorting Networks Resistant to  $k$  Passive Faults in *Proc. 7th SIAM Symposium on Discrete Algorithms* (1996), 242-251 (also accepted for *SIAM J. Comput.*).
5. M. Schimmler, C. Starke, A Correction Network for  $N$ -Sorters, *SIAM J. Comput.* 18 (1989), 1179-1197.
6. A.C. Yao, Bounds on Selection Networks, *SIAM J. Comput.* 9 (1980), 566-582.
7. A.C. Yao, F.F. Yao, On Fault-Tolerant Networks for Sorting, *SIAM J. Comput.* 14 (1985), 120-128.

# Least Adaptive Optimal Search with Unreliable Tests

Ferdinando Cicalese<sup>1,\*</sup>, Daniele Mundici<sup>2</sup>, and Ugo Vaccaro<sup>1</sup>

<sup>1</sup> Dipartimento di Informatica ed Applicazioni, University of Salerno,  
84081 Baronissi (SA), Italy  
[{cicalese,uv}@dia.unisa.it,](mailto:{cicalese,uv}@dia.unisa.it)  
<http://www.dia.unisa.it/{~cicalese,~uv}>

<sup>2</sup> Dipartimento Scienze Informazione, University of Milan,  
Via Comelico 39-41, 20135 Milan, Italy  
[mundici@mailserver.unimi.it](mailto:mundici@mailserver.unimi.it)

**Abstract.** We consider the basic problem of searching for an unknown  $m$ -bit number by asking the minimum possible number of yes-no questions, when up to a finite number  $e$  of the answers may be erroneous. In case the  $(i + 1)$ th question is adaptively asked after receiving the answer to the  $i$ th question, the problem was posed by Ulam and Rényi and is strictly related to Berlekamp's theory of error correcting communication with noiseless feedback. Conversely, in the fully non-adaptive model when *all* questions are asked before knowing *any* answer, the problem amounts to finding a shortest  $e$ -error correcting code. Let  $q_e(m)$  be the smallest integer  $q$  satisfying Berlekamp's bound  $\sum_{i=0}^e \binom{q}{i} \leq 2^{q-m}$ . Then at least  $q_e(m)$  questions are necessary, in the adaptive, as well as in the non-adaptive model. In the fully adaptive case, optimal searching strategies using exactly  $q_e(m)$  questions always exist *up to finitely many exceptional  $m$ 's*. At the opposite non-adaptive case, searching strategies with exactly  $q_e(m)$  questions—or equivalently, perfect  $e$ -error correcting codes with  $2^m$  codewords of length  $q_e(m)$ —are rather the exception, already for  $e = 2$ , and do not exist for  $e > 2$ . In this paper we show that for *any*  $e > 1$  and sufficiently large  $m$ , optimal—indeed, perfect—strategies do exist using *a first batch of  $m$  non-adaptive questions* and then, only depending on the answers to these  $m$  questions, *a second batch of  $q_e(m) - m$  non-adaptive questions*. Since even in the fully adaptive case,  $q_e(m) - 1$  questions do not suffice to find the unknown number, and  $q_e(m)$  questions generally do not suffice in the non-adaptive case, the results of our paper provide  $e$ -fault tolerant searching strategies with minimum adaptiveness and minimum number of tests.

## 1 Introduction

We consider the following scenario: Two players, called Questioner and Responder, first agree on fixing an integer  $m$  and a search space  $S = \{0, \dots, 2^m - 1\}$ .

---

\* Partially supported by ENEA

Then the Responder thinks of a number  $x \in S$  and the Questioner must find out  $x$  by asking questions to which the Responder can only answer yes or no. It is agreed that the Responder is allowed to lie (or just to be inaccurate) at most  $e$  times, where the integer  $e$  is fixed and known to the Questioner. We are interested in the problem of determining the minimum number of questions the Questioner has to ask in order to infallibly guess the number  $x$ .

When the questions are asked *adaptively*, i.e., the  $i$ th question is asked knowing the answer to the  $(i-1)$ th question, the problem is generally referred to as the Ulam-Rényi game, [29, p. 281], [24, p. 47], and is strictly related to Berlekamp's theory of error correcting communication with noiseless feedback [6]. At the other, *non-adaptive* extreme, when the totality of questions is asked at the outset, before knowing *any* answer, the problem amounts to finding a shortest  $e$ -error correcting binary code with  $2^m$  codewords.

It is known that at least  $q_e(m)$  questions are *necessary* in the adaptive and, a fortiori, in the non-adaptive case—where  $q_e(m)$  is the smallest integer  $q$  satisfying Berlekamp's bound  $\sum_{i=0}^e \binom{q}{i} \leq 2^{q-m}$ . In the fully adaptive case, an important result of Spencer [26] shows that  $q_e(m)$  questions are always sufficient, up to finitely many exceptional  $m$ 's. Optimal searching strategies had been previously exhibited by [22], [11], [21], respectively for the case  $e = 1$ ,  $e = 2$  and  $e = 3$ . Thus, fully adaptive fault tolerant search can be performed in a very satisfactory manner.

However, in many practical situations it is desirable to have searching strategies with “small degree” of adaptiveness, that is, searching strategies in which all questions (or at least, many of them) can be prepared in advance, and asked in parallel. This is the case, e.g., when the Questioner and the Responder are far away from each other and can interact only on a slow channel; or in all situations when formulating the queries is a costly process, and therefore the Questioner finds it more convenient and time-saving to prepare them in advance. We refer to the monographs [313] for a discussion on the power of adaptive and non-adaptive searching strategies and their possible uses in different contexts.

Unfortunately, in the totally non-adaptive case, a series of negative results culminating in the celebrated paper by Tietäväinen [28] (also see [17]) shows that searching strategies with exactly  $q_e(m)$  questions—or equivalently, perfect binary  $e$ -errors correcting codes with  $2^m$  codewords of length  $q_e(m)$ —are sporadic exceptions already for  $e = 2$ , and do not exist for  $e > 2$ , except in trivial cases. Thus, adaptiveness in Ulam-Rényi games can be completely eliminated *only by significantly increasing the number of questions in the solution strategy*<sup>1</sup>. Our purpose in this paper is to investigate the minimum amount of adaptiveness required by all successful searching strategies with exactly  $q_e(m)$  questions.

---

<sup>1</sup> The situation is completely different in the case of no lies: here an optimal, totally non-adaptive searching strategy with  $\lceil \log |S| \rceil$  questions simply amounts to asking  $\lceil \log |S| \rceil$  queries about the locations of the bit 1 in the binary expansion of the unknown number  $x \in S$ .

## 1.1 Our Results

We exactly quantify the minimum amount of adaptiveness needed to solve the Ulam-Renyi problem, while still constraining the total number of questions to Berlekamp’s minimum  $q_e(m)$ . Our main result is that for each  $e$ , and for all sufficiently large  $m$ , there exist searching strategies of shortest length (using *exactly* the minimum number  $q_e(m)$  of questions) in which questions can be submitted to the Responder in *only two* rounds. Specifically, for the Questioner to infallibly guess the Responder’s secret number  $x \in S$  it is *sufficient* to ask a first batch of  $m$  non-adaptive questions, and then, only depending on the  $m$ -tuple of answers, ask a second mini-batch of  $n$  non-adaptive questions. Our strategies are *perfect*, in that  $m + n$  coincides with Berlekamp’s minimum  $q_e(m)$ , the number of questions that are a priori *necessary* to accommodate all possible answering strategies of the Responder—once he is allowed to lie up to  $e$  times. Since the Questioner can adapt his strategy only once, our paper yields  $e$ -fault tolerant search strategies with *minimum* adaptiveness and the least possible number of tests. Our main tool is the discovery of a close relation between searching strategies tolerating  $e$  lies and certain special families of error correcting codes, which will be described in Section 3. In the last section we specialize our analysis to the case  $e = 3$ ; we shall give an explicit description of our searching strategies for the Ulam-Renyi game, for all  $m \geq 99$ .

## 1.2 Related Work

The general issue of coping with unreliable information (and/or unreliable components) in computing is an important problem in computer science, and its study goes back to the work of von Neumann [30]. The problem of dealing with erroneous information in search strategies (what we call here Ulam-Renyi game) has received considerable attention in the last decades, beginning with [25] (see [2, 4, 5, 9, 11, 12, 20, 22, 26] and references therein). The survey paper [14] gives a detailed account of the relevant literature on the subject. In the paper [15] the Ulam-Renyi game is embedded in a broader context.

We have already mentioned the connections between Ulam-Renyi games and Berlekamp’s theory of error correcting communication with noiseless feedback [6]. Other interesting connections between Ulam-Renyi games and different areas of computer science and logic have also been found (see for instance [8, 18]). For the sake of conciseness, we shall limit ourselves to mentioning here only those results which are directly related to our present issue of adaptive vs. non-adaptive search. It is well known that for  $e = 1$ , Hamming codes yield non-adaptive searching strategies (i.e., one round strategies) with the smallest possible number  $q_1(m)$  of questions—indeed, Pelc [23] showed that adaptiveness in this case is irrelevant even under the stronger assumption that repetition of the same question is forbidden. The first significant case where the dichotomy between adaptive and non-adaptive search makes its appearance is when  $e = 2$ . Two-round optimal strategies for the case  $e = 2$  were given in [10]. Our paper extends the result of [10] to the case of an *arbitrary* number  $e$  of errors/lies. Other results related

to the issue of *fully* adaptive vs. *totally non*-adaptive searching strategies, are contained in [12,27].

## 2 The Ulam-Renyi Game

For some fixed integer  $m$ , let  $S = \{0, 1, \dots, 2^m - 1\}$  be the search space. By a *yes-no question* we simply mean an arbitrary subset  $T$  of  $S$ . If the answer to the question  $T$  is “yes”, numbers in  $T$  are said to *satisfy* the answer, while numbers in  $S \setminus T$  *falsify* it. A negative answer to question  $T$  has the same effect as a positive answer to the opposite question  $S \setminus T$ . At any stage of the game, a number  $y \in S$  must be rejected from consideration if, and only if, it falsifies more than  $e$  answers. The remaining numbers of  $S$  still are possible candidates for the unknown  $x$ . At any time the Questioner’s *state* of knowledge is represented by an  $e$ -tuple  $\sigma = (A_0, A_1, A_2, \dots, A_e)$  of pairwise disjoint subsets of  $S$ , where  $A_i$  is the set of numbers falsifying exactly  $i$  answers,  $i = 0, 1, 2, \dots, e$ . The *initial state* is naturally given by  $(S, \emptyset, \emptyset, \dots, \emptyset)$ . A state  $(A_0, A_1, A_2, \dots, A_e)$  is *final* iff  $A_0 \cup A_1 \cup A_2 \cup \dots \cup A_e$  either has exactly one element, or is empty. In this latter case, evidently, more than  $e$  lies have been told.

For any state  $\sigma = (A_0, A_1, A_2, \dots, A_e)$  and question  $T \subseteq S$ , the two states  $\sigma^{yes}$  and  $\sigma^{no}$  respectively resulting from a positive or a negative answer, are given by

$$\sigma^{yes} = (A_0^{yes}, A_1^{yes}, \dots, A_e^{yes}) \quad \text{and} \quad \sigma^{no} = (A_0^{no}, A_1^{no}, \dots, A_e^{no}) \quad (1)$$

where, for the sake of definiteness, we let  $A_{-1} = \emptyset$ , and

$$A_i^{yes} = (A_i \cap T) \cup (A_{i-1} \setminus T) \quad \text{and} \quad A_i^{no} = (A_i \setminus T) \cup (A_{i-1} \cap T) \quad (2)$$

for each  $i = 0, 1, \dots, e$ . Given a state  $\sigma$ , suppose questions  $T_1, \dots, T_t$  have been asked and answers  $\mathbf{b} = b_1, \dots, b_t$  have been received (with  $b_i \in \{\text{yes, no}\}$ ). Iterated application of the above formulas yields a sequence of states

$$\sigma_0 = \sigma, \quad \sigma_1 = \sigma_0^{b_1}, \quad \sigma_2 = \sigma_1^{b_2}, \quad \dots, \quad \sigma_t = \sigma_{t-1}^{b_t}. \quad (3)$$

By a *strategy*  $\mathcal{S}$  with  $q$  *questions* we mean the binary tree of depth  $q$ , where each node  $\nu$  is mapped into a question  $T_\nu$ , and the two edges  $\eta_{\text{left}}, \eta_{\text{right}}$  generated by  $\nu$  are respectively labelled *yes* and *no*. Let  $\boldsymbol{\eta} = \eta_1, \dots, \eta_q$  be a path in  $\mathcal{S}$ , from the root to a leaf, with respective labels  $b_1, \dots, b_q$ , generating nodes  $\nu_1, \dots, \nu_q$  and associated questions  $T_{\nu_1}, \dots, T_{\nu_q}$ . Fix an arbitrary state  $\sigma$ . Then, according to (3), iterated application of (1)-(2) naturally transforms  $\sigma$  into  $\sigma^\eta$  (where the dependence on the  $b_j$  and  $T_j$  is understood). We say that strategy  $\mathcal{S}$  is *winning* for  $\sigma$  iff for every path  $\boldsymbol{\eta}$  the state  $\sigma^\eta$  is final. A strategy is said to be *non-adaptive* iff all nodes at the same depth of the tree are mapped into the same question.

Let  $\sigma = (A_0, A_1, A_2, \dots, A_e)$  be a state. For each  $i = 0, 1, 2, \dots, e$  let  $a_i = |A_i|$  be the number of elements of  $A_i$ . Then the  $e$ -tuple  $(a_0, a_1, a_2, \dots, a_e)$  is called

the *type* of  $\sigma$ . The *Berlekamp weight* of  $\sigma$  before  $q$  questions,  $q = 0, 1, 2, \dots$ , is given by

$$w_q(\sigma) = \sum_{i=0}^e a_i \sum_{j=0}^{e-i} \binom{q}{j}. \quad (4)$$

The *character*  $\text{ch}(\sigma)$  of a state  $\sigma$  is the smallest integer  $q \geq 0$  such that  $w_q(\sigma) \leq 2^q$ .

By abuse of notation, the weight of *any* state  $\sigma$  of type  $(a_0, a_1, a_2, \dots, a_e)$  before  $q$  questions will be denoted  $w_q(a_0, a_1, a_2, \dots, a_e)$ . Similarly, its character will also be denoted  $\text{ch}(a_0, a_1, a_2, \dots, a_e)$ .

As an immediate consequence of the above definition we have the following monotonicity properties: For any two states  $\sigma' = (A'_0, A'_1, A'_2, \dots, A'_e)$  and  $\sigma'' = (A''_0, A''_1, A''_2, \dots, A''_e)$  respectively of type  $(a'_0, a'_1, a'_2, \dots, a'_e)$  and  $(a''_0, a''_1, a''_2, \dots, a''_e)$ , if  $a'_i \leq a''_i$  for all  $i = 0, 1, 2, \dots, e$  then

$$\text{ch}(\sigma') \leq \text{ch}(\sigma'') \text{ and } w_q(\sigma') \leq w_q(\sigma'') \quad (5)$$

for each  $q \geq 0$ . Moreover, if there exists a winning strategy for  $\sigma''$  with  $q$  questions then there exists also a winning strategy for  $\sigma'$  with  $q$  questions [6]. Note that  $\text{ch}(\sigma) = 0$  iff  $\sigma$  is a final state.

**Lemma 1.** [6] *Let  $\sigma$  be an arbitrary state, and  $T \subseteq S$  a question. Let  $\sigma^{\text{yes}}$  and  $\sigma^{\text{no}}$  be as in (1)-(2).*

- (i) (Conservation Law). *For any integer  $q \geq 1$  we have  $w_q(\sigma) = w_{q-1}(\sigma^{\text{yes}}) + w_{q-1}(\sigma^{\text{no}})$ .*
- (ii) (Berlekamp's lower bound). *If  $\sigma$  has a winning strategy with  $q$  questions then  $q \geq \text{ch}(\sigma)$ .*  $\square$

In complete analogy with the notion of perfect error correcting code [17], we say that a winning strategy for  $\sigma$  with  $q$  questions is *perfect* iff  $q = \text{ch}(\sigma)$ . In agreement with the above notation, we shall write  $q_e(m)$  instead of  $\text{ch}(2^m, 0, \dots, 0)$ .

Let  $\sigma = (A_0, A_1, A_2, \dots, A_e)$  be a state. Let  $T \subseteq S$  be a question. We say that  $T$  is *balanced for  $\sigma$*  iff for each  $j = 0, 1, 2, \dots, e$ , we have  $|A_j \cap T| = |A_j \setminus T|$ . The following is easy to prove.

**Lemma 2.** *Let  $T$  be a balanced question for a state  $\sigma = (A_0, A_1, A_2, \dots, A_e)$ . Let  $n = \text{ch}(\sigma)$ . Let  $\sigma^{\text{yes}}$  and  $\sigma^{\text{no}}$  be as in (1)-(2) above. Then*

- (i)  $w_q(\sigma^{\text{yes}}) = w_q(\sigma^{\text{no}})$ , for each integer  $q \geq 0$ ,
- (ii)  $\text{ch}(\sigma^{\text{yes}}) = \text{ch}(\sigma^{\text{no}}) = n - 1$ .

### 3 Strategies vs. Codes

Let us first remind some notations from Coding Theory, for more see [17].

Fix an integer  $n > 0$  and let  $\mathbf{x}, \mathbf{y} \in \{0, 1\}^n$ . The *Hamming distance*  $d_H(\mathbf{x}, \mathbf{y})$  is defined by

$$d_H(\mathbf{x}, \mathbf{y}) = |\{i \in \{1, \dots, n\} \mid x_i \neq y_i\}|,$$

where, as above,  $|A|$  denotes the number of elements of  $A$ , and  $x_i$  (resp.  $y_i$ ) denotes the  $i$ th components of  $\mathbf{x}$  (resp.  $\mathbf{y}$ ).

The *Hamming sphere*  $\mathcal{B}_r(\mathbf{x})$  with radius  $r$  and center  $\mathbf{x}$  is the set of elements of  $\{0, 1\}^n$  whose Hamming distance from  $\mathbf{x}$  is at most  $r$ , in symbols,

$$\mathcal{B}_r(\mathbf{x}) = \{\mathbf{y} \in \{0, 1\}^n \mid d_H(\mathbf{x}, \mathbf{y}) \leq r\}.$$

Notice that for any  $\mathbf{x} \in \{0, 1\}^n$ , and  $r \geq 0$ , we have  $|\mathcal{B}_r(\mathbf{x})| = \sum_{i=0}^r \binom{n}{i}$ . The *Hamming weight*  $w_H(\mathbf{x})$  of  $\mathbf{x}$  is the number of non-zero digits of  $\mathbf{x}$ . Throughout this paper, by a code we shall mean a binary code, in the following sense:

**Definition 1.** A (binary) code  $\mathcal{C}$  of length  $n$  is a non-empty subset of  $\{0, 1\}^n$ . Its elements are called codewords. The minimum distance of  $\mathcal{C}$  is given by

$$\delta(\mathcal{C}) = \min\{d_H(\mathbf{x}, \mathbf{y}) \mid \mathbf{x}, \mathbf{y} \in \mathcal{C}, \mathbf{x} \neq \mathbf{y}\}.$$

We say that  $\mathcal{C}$  is an  $(n, m, d)$  code iff  $\mathcal{C}$  has length  $n$ ,  $|\mathcal{C}| = m$  and  $\delta(\mathcal{C}) = d$ . The minimum weight of  $\mathcal{C}$  is the minimum of the Hamming weights of its codewords, in symbols,

$$\mu(\mathcal{C}) = \min\{w_H(\mathbf{x}) \mid \mathbf{x} \in \mathcal{C}\}.$$

Let  $\mathcal{C}_1$  and  $\mathcal{C}_2$  be two codes of length  $n$ . The minimum distance between  $\mathcal{C}_1$  and  $\mathcal{C}_2$  is defined by

$$\Delta(\mathcal{C}_1, \mathcal{C}_2) = \min\{d_H(\mathbf{x}, \mathbf{y}) \mid \mathbf{x} \in \mathcal{C}_1, \mathbf{y} \in \mathcal{C}_2\}.$$

We now describe a correspondence between non-adaptive winning strategies and certain special codes. This will be a key tool to prove the main results of our paper.

**Lemma 3.** Let  $\sigma = (A_0, A_1, A_2, \dots, A_e)$  be a state of type  $(a_0, a_1, a_2, \dots, a_e)$ . Let  $n \geq \text{ch}(\sigma)$ . Then a non-adaptive winning strategy for  $\sigma$  with  $n$  questions exists if and only if for all  $i = 0, 1, 2, \dots, e-1$  there are integers  $d_i \geq 2(e-i)+1$ , together with an  $e$ -tuple of codes  $\Gamma = \{\mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_{e-1}\}$ , such that each  $\mathcal{C}_i$  is an  $(n, a_i, d_i)$  code, and  $\Delta(\mathcal{C}_i, \mathcal{C}_j) \geq 2e-(i+j)+1$ , (whenever  $0 \leq i < j \leq e-1$ ).

*Proof.* We first prove the implication *strategy*  $\Rightarrow$  *codes*.

Assume  $\sigma = (A_0, A_1, A_2, \dots, A_e)$  to be a state of type  $(a_0, a_1, a_2, \dots, a_e)$  having a non-adaptive winning strategy  $\mathcal{S}$  with  $n$  questions  $T_1, \dots, T_n$ ,  $n \geq \text{ch}(\sigma)$ . Let the map

$$z \in A_0 \cup A_1 \cup A_2 \cup \dots \cup A_e \mapsto \mathbf{z}^{\mathcal{S}} \in \{0, 1\}^n$$

send each  $z \in A_0 \cup A_1 \cup A_2 \cup \dots \cup A_e$  into the  $n$ -tuple of bits  $\mathbf{z}^{\mathcal{S}} = z_1^{\mathcal{S}} \dots z_n^{\mathcal{S}}$  arising from the sequence of “true” answers to the questions “does  $z$  belong to  $T_1$ ?”, “does  $z$  belong to  $T_2$ ?”, ..., “does  $z$  belong to  $T_n$ ?”, via the identifications

$1 = \text{yes}$ ,  $0 = \text{no}$ . More precisely, for each  $j = 1, \dots, n$ ,  $z_j^S = 1$  iff  $z \in T_j$ . Let  $\mathcal{C} \subseteq \{0, 1\}^n$  be the range of the map  $z \mapsto z^S$ . We shall first prove that, for every  $i = 0, \dots, e - 1$ , there exists an integer  $d_i \geq 2(e - i) + 1$  such that the set  $\mathcal{C}_i = \{\mathbf{y}^S \in \mathcal{C} \mid y \in A_i\}$  is an  $(n, a_i, d_i)$  code.

Since  $\mathcal{S}$  is winning, the map  $z \mapsto z^S$  is one-to-one, whence in particular  $|\mathcal{C}_i| = a_i$ , for any  $i = 0, 1, 2, \dots, e - 1$ . Moreover by definition, the  $\mathcal{C}_i$ 's are subsets of  $\{0, 1\}^n$ .

*Claim 1.*  $\delta(\mathcal{C}_i) \geq 2(e - i) + 1$ , for  $i = 0, \dots, e - 1$ .

For otherwise (absurdum hypothesis) assuming  $c$  and  $d$  to be two distinct elements of  $A_i$  such that  $d_H(\mathbf{c}^S, \mathbf{d}^S) \leq 2(e - i)$ , we will prove that  $\mathcal{S}$  is not a winning strategy. We can safely assume  $c_j^S = d_j^S$  for each  $j = 1, \dots, n - 2(e - i)$ . Suppose the answer to question  $T_j$  is “yes” or “no” according as  $c_j^S = 1$  or  $c_j^S = 0$ , respectively. Then after  $n - 2(e - i)$  answers, the resulting state has the form  $\sigma' = (A'_0, \dots, A'_i, \dots, A'_e)$ , with  $\{c, d\} \subseteq A'_i$ , whence the type of  $\sigma'$  is  $(a'_0, \dots, a'_i, \dots, a'_e)$  with  $a'_i \geq 2$ . Since by [6, Lemma 2.5],  $\text{ch}(\sigma') \geq \text{ch}(0, 0, \dots, 0, 2, 0, \dots, 0) = 2(e - i) + 1$  then from Lemma II(ii) it follows that the remaining  $2(e - i)$  questions/answers do not suffice to reach a final state, thus contradicting the assumption that  $\mathcal{S}$  is winning.

*Claim 2.* For any  $0 \leq i < j \leq e - 1$  and for each  $y \in A_i$  and  $h \in A_j$  we have the inequality  $d_H(\mathbf{y}^S, \mathbf{h}^S) \geq 2e - (i + j) + 1$ .

For otherwise (absurdum hypothesis) let  $y \in A_i, h \in A_j$  be a counterexample, and  $d_H(\mathbf{y}^S, \mathbf{h}^S) \leq 2e - (i + j)$ . Writing  $\mathbf{y}^S = y_1^S \dots y_n^S$  and  $\mathbf{h}^S = h_1^S \dots h_n^S$ , it is no loss of generality to assume  $h_k^S = y_k^S$ , for all  $k = 1, \dots, n - (2e - (i + j))$ . Suppose that the answer to question  $T_k$  is “yes” or “no” according as  $h_k^S = 1$  or  $h_k^S = 0$ , respectively. Then the state resulting from these answers has the form  $\sigma'' = (A''_0, A''_1, A''_2, \dots, A''_e)$ , where  $y \in A''_i$  and  $h \in A''_j$ . Since by [6, Lemma 2.5],  $\text{ch}(\sigma'') \geq \text{ch}(0, \dots, 0, 1, 0, \dots, 0, 1, 0, \dots, 0) = 2e - (i + j) + 1$ , then Lemma II(ii) again shows that  $2e - (i + j)$  additional questions will not suffice to find the unknown number. This contradicts the assumption that  $\mathcal{S}$  is a winning strategy.

In conclusion, for all  $i = 0, 1, \dots, e - 1$ ,  $\mathcal{C}_i$  is an  $(n, a_i, d_i)$  code with  $d_i \geq 2(e - i) + 1$  and for all  $j = 0, \dots, i - 1, i + 1, \dots, e - 1$ , we have the desired inequality  $\Delta(\mathcal{C}_i, \mathcal{C}_j) \geq 2e - (i + j) + 1$ .

Now we prove the converse implication: *strategy  $\Leftarrow$  codes*.

Let  $\Gamma = \{\mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_{e-1}\}$  be a family of codes satisfying the hypothesis. Let

$$\mathcal{H} = \bigcup_{i=0}^{e-1} \bigcup_{\mathbf{x} \in \mathcal{C}_i} \mathcal{B}_{e-i}(\mathbf{x}).$$

By hypothesis, for any  $i, j \in \{0, 1, \dots, e - 1\}$  and  $\mathbf{x} \in \mathcal{C}_i, \mathbf{y} \in \mathcal{C}_j$  we have  $d_H(\mathbf{x}, \mathbf{y}) \geq 2e - (i + j) + 1$ . It follows that the Hamming spheres  $\mathcal{B}_{e-i}(\mathbf{x}), \mathcal{B}_{e-j}(\mathbf{y})$  are pairwise disjoint and hence

$$|\mathcal{H}| = \sum_{i=0}^{e-1} a_i \sum_{j=0}^{e-i} \binom{n}{j}. \quad (6)$$

Let  $\mathcal{D} = \{0, 1\}^n \setminus \mathcal{H}$ . Since  $n \geq \text{ch}(a_0, a_1, a_2, \dots, a_e)$ , by definition of character we have  $2^n \geq \sum_{i=0}^e a_i \sum_{j=0}^{e-i} \binom{n}{j}$ . From (6) it follows that

$$|\mathcal{D}| = 2^n - \sum_{i=0}^{e-1} a_i \sum_{j=0}^{e-i} \binom{n}{j} \geq a_e. \quad (7)$$

Let  $\sigma = (A_0, A_1, A_2, \dots, A_e)$  be an arbitrary state of type  $(a_0, a_1, a_2, \dots, a_e)$ . Let us now fix, once and for all,  $e+1$  one-one maps  $f_i: A_i \rightarrow \mathcal{C}_i$ , for  $i = 0, 1, \dots, e-1$  and  $f_e: A_e \rightarrow \mathcal{D}$ . The existence of the map  $f_i$ , for all  $i = 0, 1, \dots, e$ , is ensured by our assumptions about  $\Gamma$ , together with (7).

Let the map  $f: A_0 \cup A_1 \cup A_2 \cup \dots \cup A_e \rightarrow \{0, 1\}^n$  be defined by cases as follows:

$$f(y) = \begin{cases} f_0(y), & y \in A_0 \\ f_1(y), & y \in A_1 \\ \vdots \\ f_e(y), & y \in A_e \end{cases} \quad (8)$$

Note that  $f$  is one-one. For each  $y \in A_0 \cup A_1 \cup A_2 \cup \dots \cup A_e$  and  $j = 1, \dots, n$  let  $f(y)_j$  be the  $j$ th bit of the binary vector corresponding to  $y$  via  $f$ . We can now exhibit the questions  $T_j$  of our searching strategies:

For each  $j = 1, \dots, n$  let the set  $T_j \subseteq S$  be defined by  $T_j = \{z \in \bigcup_{i=0}^e A_i \mid f(z)_j = 1\}$ . Intuitively, letting  $x_*$  denote the unknown number,  $T_j$  asks “is the  $j$ th bit of  $f(x_*)$  equal to one?”

Again writing  $yes = 1$  and  $no = 0$ , the answers to questions  $T_1, \dots, T_n$  determine an  $n$ -tuple of bits  $\mathbf{b} = b_1 \cdots b_n$ . We shall show that the sequence  $T_1, \dots, T_n$  yields an optimal non-adaptive winning strategy for  $\sigma$ . Let  $\sigma_1 = \sigma^{b_1}$ ,  $\sigma_2 = \sigma_1^{b_2}, \dots, \sigma_n = \sigma_{n-1}^{b_n}$ . Arguing by cases we shall show that  $\sigma_n = (A_0^*, A_1^*, \dots, A_e^*)$  is a final state.

By (1)-(2), for all  $i = 0, 1, \dots, e$ , any  $z \in A_{e-i}$  that falsifies  $> i$  answers does not survive in  $\sigma_n$ —in the sense that  $z \notin A_0^* \cup A_1^* \cup \dots \cup A_e^*$ .

*Case 1.*  $\mathbf{b} \notin \bigcup_{i=0}^e \bigcup_{y \in A_i} \mathcal{B}_{e-i}(f(y))$ .

For all  $i = 0, 1, \dots, e$ , and for each  $y \in A_i$  we must have  $y \notin A_0^* \cup A_1^* \cup \dots \cup A_e^*$ . Indeed, the assumption  $\mathbf{b} \notin \mathcal{B}_{e-i}(f(y))$  implies  $d_H(f(y), \mathbf{b}) > e - i$ , whence  $y$  falsifies  $> e - i$  of the answers to  $T_1, \dots, T_n$ , and  $y$  does not survive in  $\sigma_n$ . We have proved that  $A_0^* \cup A_1^* \cup \dots \cup A_e^*$  is empty, and  $\sigma_n$  is a final state.

*Case 2.*  $\mathbf{b} \in \mathcal{B}_{e-i}(f(y))$  for some  $i \in \{0, 1, \dots, e\}$  and  $y \in A_i$ .

Then  $y \in A_0^* \cup A_1^* \cup \dots \cup A_e^*$ , because  $d_H(f(y), \mathbf{b}) \leq e - i$ , whence  $y$  falsifies  $\leq e - i$  answers. Our assumptions about  $\Gamma$  ensure that, for all  $j = 0, 1, \dots, e$  and for all  $y' \in A_j$  and  $y \neq y'$ , we have  $\mathbf{b} \notin \mathcal{B}_{e-j}(f(y'))$ . Thus,  $d_H(f(y'), \mathbf{b}) > e - j$  and  $y'$  falsifies  $> e - j$  of the answers to  $T_1, \dots, T_n$ , whence  $y'$  does not survive in  $\sigma_n$ . This shows that for any  $y' \neq y$ ,  $y' \notin A_0^* \cup A_1^* \cup \dots \cup A_e^*$ . Therefore,  $A_0^* \cup A_1^* \cup \dots \cup A_e^*$  only contains the element  $y$ , and  $\sigma_n$  is a final state.

## 4 Optimal Strategies with Minimum Adaptiveness

### 4.1 The First Batch of Questions

Recall that  $q_e(m) = \text{ch}(2^m, 0, \dots, 0)$  is the smallest integer  $q \geq 0$  such that  $2^q \geq 2^m \left(\binom{q}{e} + \binom{q}{e-1} + \dots + \binom{q}{2} + q + 1\right)$ . By Lemma 1(ii), at least  $q_e(m)$  questions are *necessary* to guess the unknown number  $x_* \in S = \{0, 1, \dots, 2^m - 1\}$ , if up to  $e$  answers may be erroneous. The aim of the rest of this paper is to prove that, conversely, for sufficiently large  $m$ ,  $q_e(m)$  questions are *sufficient* under the following constraint: first we use a predetermined non-adaptive batch of  $m$  questions  $D_1, \dots, D_m$ , and then, only depending on the answers, we ask the remaining  $q_e(m) - m$  questions in a second non-adaptive batch. The *first batch of questions* is easily described as follows:

For each  $i = 1, 2, \dots, m$ , let  $D_i \subseteq S$  denote the question “Is the  $i$ th binary digit of  $x_*$  equal to 1?” Thus a number  $y \in S$  belongs to  $D_i$  iff the  $i$ th bit  $y_i$  of its binary expansion  $\mathbf{y} = y_1 \dots y_m$  is equal to 1.

Upon identifying  $1 = \text{yes}$  and  $0 = \text{no}$ , let  $b_i \in \{0, 1\}$  be the answer to question  $D_i$ . Let  $\mathbf{b} = b_1 \dots b_m$ . Repeated applications of (1)-(2) beginning with the initial state  $\sigma = (S, \emptyset, \dots, \emptyset)$ , shows that the resulting state as an effect of the answers  $b_1 \dots b_m$ , is an  $(e+1)$ -tuple  $\sigma^\mathbf{b} = (A_0, A_1, \dots, A_e)$ , where

$$A_i = \{y \in S \mid d_H(\mathbf{y}, \mathbf{b}) = i\} \quad \text{for all } i = 0, 1, \dots, e.$$

Direct verification yields

$$|A_0| = 1, \quad |A_1| = m, \dots, |A_e| = \binom{m}{e}.$$

Thus  $\sigma^\mathbf{b}$  has type  $(1, m, \binom{m}{2}, \dots, \binom{m}{e})$ . As in (3), let  $\sigma_i$  be the state resulting after the first  $i$  answers, beginning with  $\sigma_0 = \sigma$ . Since each question  $D_i$  is balanced for  $\sigma_{i-1}$ , an easy induction using Lemma 2 yields  $\text{ch}(\sigma^\mathbf{b}) = q_e(m) - m$ .

For each  $m$ -tuple  $\mathbf{b} \in \{0, 1\}^m$  of possible answers, we shall construct a non-adaptive strategy  $S_\mathbf{b}$  with  $\text{ch}(1, m, \binom{m}{2}, \dots, \binom{m}{e})$  questions, which turns out to be winning for the state  $\sigma^\mathbf{b}$ . To this purpose, let us consider the values of  $\text{ch}(1, m, \binom{m}{2}, \dots, \binom{m}{e})$  for  $m \geq 1$ .

**Definition 2.** Let  $e \geq 0$  and  $n \geq 2e$  be arbitrary integers. The critical index  $m_{n,e}$  is the largest integer  $m \geq 0$  such that  $\text{ch}(1, m, \binom{m}{2}, \dots, \binom{m}{e}) = n$ .

**Lemma 4.** Let  $e \geq 0$  and  $n \geq 2e$  be arbitrary integers. Then  $m_{n,e} < \sqrt[e]{e!} 2^{\frac{n}{e}} + e$ .

*Proof.* Recall that  $m_{n,e} = \max \{m \mid w_n(1, m, \binom{m}{2}, \dots, \binom{m}{e}) \leq 2^n\}$ . We now set  $m^* = \sqrt[e]{e!} 2^{\frac{n}{e}} + e$ . Then, the desired result now directly follows from the inequality  $w_n(1, m^*, \binom{m^*}{2}, \dots, \binom{m^*}{e}) > 2^n$ . Indeed, we have

$$\begin{aligned}
w_n \left( 1, m^*, \binom{m^*}{2}, \dots, \binom{m^*}{e} \right) &> w_n \left( 0, 0, \dots, 0, \binom{m^*}{e} \right) \\
&= \binom{m^*}{e} = \frac{m^*(m^*-1)\cdots(m^*-e+1)}{e!} \\
&\geq \frac{\left(\sqrt[e]{e!} 2^{n/e}\right)^e}{e!} = 2^n.
\end{aligned}$$

## 4.2 The Second Batch of Questions

We now prove that for all sufficiently large  $m$  there exists a second batch of  $n = q_e(m) = \text{ch}(1, m, \binom{m}{2}, \dots, \binom{m}{e})$  non-adaptive questions allowing the Questioner to infallibly guess the Responder's secret number. We first need the following lemma.

**Lemma 5.** *For any fixed  $e$  and all sufficiently large  $n$  there exists a family of codes  $\Gamma = \{\mathcal{C}_0, \mathcal{C}_1, \dots, \mathcal{C}_{e-1}\}$  together with integers  $d_i \geq 2(e-i)+1$  ( $i = 0, 1, \dots, e-1$ ) such that*

- (i) *Each  $\mathcal{C}_i$  is an  $(n, \binom{m_{n,e}}{i}, d_i)$  code;*
- (ii)  *$\Delta(\mathcal{C}_i, \mathcal{C}_j) \geq 2e - (i+j) + 1$ , (whenever  $0 \leq i < j \leq e-1$ .)*

*Proof.* Let  $n' = n - e^2$ . First we prove the existence of an  $(n', \binom{m_{n,e}}{e-1}, d')$  code, with  $d' = 2e + 1$ . From Lemma 4 together with the well known inequality  $e! \leq \frac{(e+1)^e}{2^e}$ , it follows that, for all sufficiently large  $n$

$$\begin{aligned}
\binom{m_{n,e}}{e-1} &< (m_{n,e})^{e-1} < (\sqrt[e]{e!} 2^{\frac{n}{e}} + e)^{e-1} \\
&\leq (e 2^{\frac{n}{e}})^{e-1} = e^{e-1} 2^{n \frac{e-1}{e}} \\
&\leq \frac{2^{n-e^2}}{\sum_{j=0}^{2e} \binom{n-e^2}{j}}.
\end{aligned}$$

The existence of the desired  $(n', \binom{m_{n,e}}{e-1}, d')$  code now follows by the well known Gilbert Bound [17].

We have proved that, for all sufficiently large  $n$ , there exists an  $(n-e^2, \binom{m_{n,e}}{e-1}, d')$  code  $\mathcal{C}'$  with  $d' \geq 2e + 1$ . For any  $i = 0, 1, \dots, e-1$  let the  $e^2$ -tuple  $\mathbf{a}_i$  be defined by

$$\mathbf{a}_i = \underbrace{00\dots0}_{(i-1)e} \underbrace{11\dots1}_e \underbrace{0\dots0}_{e^2-ie}.$$

Furthermore, let  $\mathcal{C}_i''$  be the code obtained by appending the suffix  $\mathbf{a}_i$  to the codewords of  $\mathcal{C}'$ , in symbols,

$$\mathcal{C}_i'' = \mathcal{C}' \otimes \mathbf{a}_i.$$

Trivially,  $\mathcal{C}_i''$  is an  $(n, \binom{m_{n,e}}{e-1}, 2e+1)$  code for all  $i = 0, 1, \dots, e$ . Furthermore, we have  $\Delta(\mathcal{C}_i'', \mathcal{C}_j'') = 2e \geq 2e - (i+j) + 1$ , whenever  $0 \leq i < j \leq e-1$ . For each  $i = 0, 1, \dots, e-1$ , pick a subcode  $\mathcal{C}_i \subseteq \mathcal{C}_i''$  with  $|\mathcal{C}_i| = \binom{m_{n,e}}{i}$ . Then the new family of codes  $\Gamma = \{\mathcal{C}_0, \mathcal{C}_1, \dots, \mathcal{C}_{e-1}\}$  satisfies both conditions (i) and (ii) and the proof is complete.

The following corollary implies the existence of minimum adaptiveness perfect searching strategies.

**Corollary 1.** *Fix an integer  $e \geq 0$ . Then for all sufficiently large integers  $m$  and for every state  $\sigma$  of type  $(1, m, \binom{m}{2}, \dots, \binom{m}{e})$  there exists a non-adaptive winning strategy  $\mathcal{S}$  such that the number of questions in  $\mathcal{S}$  coincides with Berlekamp's lower bound  $\text{ch}(\sigma) = q_2(m) - m$ .*

*Proof.* Let  $n = \text{ch}(\sigma)$ . By definition,  $n \rightarrow \infty$  as  $m \rightarrow \infty$ . Lemmas 5 and 3 yield a non-adaptive winning strategy with  $n$  questions for any state of type  $(1, m_{n,e}, \binom{m_{n,e}}{2}, \dots, \binom{m_{n,e}}{e})$ . By Definition 2,  $m \leq m_n$ , and a fortiori, for all sufficiently large  $m$ , a non-adaptive winning strategy with  $n$  questions exists for any state of type  $(1, m, \binom{m}{2}, \dots, \binom{m}{e})$ .

## 5 Ulam-Rényi Game with Three Lies and Minimum Adaptiveness

In this section we restrict to the particular case  $e = 3$ . We shall prove that for all  $m \geq 99$  perfect (hence, a fortiori, optimal) searching strategies do exist to find an unknown  $m$ -bit number  $x_*$  with minimum adaptiveness and up to 3 lies in the answers. States have now the form  $(A_0, A_1, A_2, A_3)$ . Proceeding as in the previous section, we may safely assume that after a first batch of  $m$  non-adaptive questions asking for the binary expansion of  $x_*$  (the *bitwise* batch), the resulting state  $\sigma$  is of type  $(1, m, \binom{m}{2}, \binom{m}{3})$  and character  $n = \text{ch}(\sigma) = q_3(m) - m$ . We shall show that a non-adaptive winning strategy for  $\sigma$  with  $n$  questions exists for each  $m \geq 99$  (we will not try to optimize this constant in the present version of this paper). We shall use the following preliminary lemma.

**Lemma 6.** *Let  $n$  and  $m$  be arbitrary integers  $\geq 1$ . For  $i = 1, 2$ , let  $\mathcal{C}_i$  be an  $(n, M_i, d_i)$  code with  $\mu(\mathcal{C}_i) \geq g_i$ , for some integers  $M_i \geq \binom{m+i-1}{i}$ ,  $d_i \geq 7 - 2i$ ,  $g_i \geq 7 - i$ . Let  $\Delta(\mathcal{C}_1, \mathcal{C}_2) \geq 4$ .*

*Then for all  $j = 1, 2, 3, \dots$ , there exists an  $(n + 3j, M', 5)$  code  $\mathcal{D}_1^{(j)}$  with  $M' \geq 2^{2j}m$ ,  $\mu(\mathcal{D}_1^{(j)}) \geq g_1$ , together with an  $(n + 3j, M'', 3)$  code  $\mathcal{D}_2^{(j)}$  such that  $M'' \geq \binom{2^{2j}m}{2}$ ,  $\mu(\mathcal{D}_2^{(j)}) \geq g_2$  and  $\Delta(\mathcal{D}_1^{(j)}, \mathcal{D}_2^{(j)}) \geq 4$ .*

*Proof.* Omitted.

**Lemma 7.** *For all  $n \geq 19$  there is an  $(n, M_1, d_1)$  code  $\mathcal{C}_{n,1}$  and an  $(n, M_2, d_2)$  code  $\mathcal{C}_{n,2}$  such that*

$$M_1 \geq m_{n,3}, \quad d_1 \geq 5, \quad M_2 \geq \binom{m_{n,3}}{2}, \quad d_2 \geq 3,$$

$$\mu(\mathcal{C}_{n,1}) \geq 6, \quad \mu(\mathcal{C}_{n,2}) \geq 5, \quad \Delta(\mathcal{C}_{n,1}, \mathcal{C}_{n,2}) \geq 4.$$

*Proof.* By direct inspection in [7, Table I-A, I-B], for  $n = 20, 21, 22$ , there exist codes  $\mathcal{D}_{n,1}, \mathcal{D}_{n,2}, \mathcal{D}_{n,3}$  such that

- (i)  $\mathcal{D}_{n,1}$  is an  $(n, M_{n,1}, 6)$  code and  $w_H(\mathbf{x}) = 6$  for any  $\mathbf{x} \in \mathcal{D}_{n,1}$ ,
- (ii)  $\mathcal{D}_{n,2}$  is an  $(n, M_{n,2}, 4)$  code and  $w_H(\mathbf{x}) = 10$  for any  $\mathbf{x} \in \mathcal{D}_{n,2}$ ,
- (iii)  $\mathcal{D}_{n,3}$  is an  $(n, M_{n,3}, 4)$  code and  $w_H(\mathbf{x}) = 13$  for any  $\mathbf{x} \in \mathcal{D}_{n,3}$ .

Moreover,

$$M_{n,1} > \sqrt[3]{6} 2^{\frac{n}{3}} + 3 \geq m_{n,3}$$

and

$$M_{n,2} + M_{n,3} > \binom{\sqrt[3]{6} 2^{\frac{n}{3}} + 4}{2} \geq \binom{m_{n,3} + 1}{2} > \binom{m_{n,3}}{2}.$$

It is apparent that  $\Delta(\mathcal{D}_{n,2}, \mathcal{D}_{n,3}) \geq 3$ . Define  $\mathcal{C}_{n,1} = \mathcal{D}_{n,1}$  and  $\mathcal{C}_{n,2} = \mathcal{D}_{n,2} \cup \mathcal{D}_{n,3}$ .

Trivially,  $\mu(\mathcal{C}_{n,1}) \geq 6$  and  $\mu(\mathcal{C}_{n,2}) \geq 5$ . Hence the claim holds for  $n = 20, 21, 22$ .

For any  $n \geq 23$ , write  $n = n' + 3j$  with  $n' \in \{20, 21, 22\}$  and  $j \geq 1$ . Then by Lemma 6 there exist an  $(n, M', 5)$  code  $\mathcal{C}_{n,1}$  with

$$M' \geq 2^j m_{n',3} > m_{n'+3j,3} = m_{n,3}$$

and an  $(n, M'', 3)$  code  $\mathcal{C}_{n,2}$  with

$$M'' \geq \binom{2^j m_{n',3}}{2} > \binom{m_{n,3}}{2}$$

such that  $\mu(\mathcal{C}_{n,1}) \geq 6$ ,  $\mu(\mathcal{C}_{n,2}) \geq 5$  and  $\Delta(\mathcal{C}_{n,1}, \mathcal{C}_{n,2}) \geq 4$ . Hence the desired result holds for all  $n \geq 20$ .

For the remaining case  $n = 19$ , direct inspection in [7, Table I-A, I-B] again yields three codes  $\mathcal{D}_{n,i}$ , ( $i = 1, 2, 3$ ) as above, with  $M_{n,1} = 172 > 127 = m_{19,3}$  and  $M_{n,2} + M_{n,3} = 8322 > 8001 = \binom{m_{19,3}}{2}$ . This concludes the proof.

**Corollary 2.** *Fix an integer  $m \geq 99$ , and let  $\sigma$  be an arbitrary state of type  $(1, m, \binom{m}{2}, \binom{m}{3})$ . Then there exists a perfect non-adaptive winning strategy  $\mathcal{S}$  for  $\sigma$  (in the sense that the number of questions in  $\mathcal{S}$  coincides with Berlekamp's lower bound  $\text{ch}(\sigma) = q_3(m) - m$ ).*

*Proof.* Let  $n = \text{ch}(\sigma)$ . From the assumption  $m \geq 99$  by direct inspection, we get  $n \geq 19$ . Lemma 7 yields an  $(n, a_1, d_1)$  code  $\mathcal{D}_1$  with  $a_1 \geq m_{n,3}$ ,  $\mu(\mathcal{D}_1) \geq 6$ ,  $d_1 \geq 5$  together with an  $(n, a_2, d_2)$  code  $\mathcal{D}_2$  with  $a_2 \geq \binom{m_{n,3}}{2}$ ,  $\mu(\mathcal{D}_2) \geq 5$ ,  $d_2 \geq 3$  satisfying the inequality  $\Delta(\mathcal{D}_1, \mathcal{D}_2) \geq 4$ . By definition,  $m \leq m_{n,3}$ . Pick subcodes  $\mathcal{C}_1 \subseteq \mathcal{D}_1$  and  $\mathcal{C}_2 \subseteq \mathcal{D}_2$  such that  $|\mathcal{C}_1| = m$  and  $|\mathcal{C}_2| = \binom{m}{2}$ . Finally let the  $(n, 1, 7)$  code  $\mathcal{C}_0$  be defined by  $\mathcal{C}_0 = \{0 \dots 0\}$ . Then the desired conclusion directly follows by Lemma 3, using the family of codes  $\Gamma = \{\mathcal{C}_0, \mathcal{C}_1, \mathcal{C}_2\}$ .

## 6 Conclusions and Open Problems

For all sufficiently large search spaces we have proved the existence of *perfect e-error* correcting search strategies where adaptiveness occurs only once. Our result is optimal in that, by Tietäväinen theorem, [28], for all  $e > 1$  adaptiveness cannot be further reduced without loosing the property of perfectness. Our results also suggest several interesting problems for future investigation.

The first problem is motivated by the *asymmetric* nature of the communication between Questioner and Responder. Indeed, in our scenario the Questioner-to-Responder channel is *noiseless*, while the channel in the opposite direction is noisy. In the cooperative model where Questioner and Responder have agreed on the searching strategy, and lies are replaced by distortions, our results show that error correction can be achieved by, first sending  $m$  bits via the noisy Responder-to-Questioner channel, then sending via the noiseless channel the  $m$ -tuple of bits actually received by the Questioner, and finally, sending to the Questioner a final tip of  $q_e(m) - m$  bits, again via the noisy channel. It seems reasonable to try to limit the use of the noiseless channel, which in practice is the more costly channel. The following problem is especially worthy of investigation: *To which extent can one decrease the number of bits sent through the noiseless channel, while still keeping to a minimum both the total number of questions and the number of non-adaptive batches of questions?* Trade-off results between the above parameters are also of interest. For general recent results on asymmetric communication channels see [1].

Following tradition, we have allowed questions to be arbitrary subsets of the search space. However, interesting research problems arise once one restricts the Questioner's expressive power. For instance, can our perfect, minimum adaptiveness, strategies be achieved by only using comparison questions and their like (as in [25,19,20])? On the other hand, which sorts of perfect minimally adaptive strategies exist in the model where the Responder is allowed a greater expressive power than mere binary answers (as, e.g., in [4,9])? It would also be of interest to extend to  $e > 3$  the non-asymptotic results of Section 5.

Finally, it would be interesting to extend our methods to other related problems in the area of computing with unreliable tests (e.g., [16]).

## References

1. M. Adler and B. Maggs, *Protocols for asymmetric communication channels*, In: Proc. of 39th IEEE FOCS, (1998).
2. J. A. Aslam and A. Dhagat, *Searching in the presence of linearly bounded errors*, In: Proceedings of the 23rd ACM STOC (1991), 486-493.
3. M. Aigner, *Combinatorial Search*, Wiley–Teubner, New York–Stuttgart, 1988.
4. M. Aigner, *Searching with lies*, J. Comb. Theory, Ser. A, **74** (1995), 43-56.
5. R. S. Borgstrom and S. Rao Kosaraju, *Comparison-based search in the presence of errors*, In: Proceedings of the 25th ACM STOC (1993), 130-136.
6. E. R. Berlekamp, *Block coding for the binary symmetric channel with noiseless, delayless feedback*, In: Error-correcting Codes, H.B. Mann (Editor), Wiley, New York (1968), 61-88.

7. A. E. Brouwer, J. B. Shearer, N.J.A. Sloane, W. D. Smith, *A New Table of Constant Weight Codes*, IEEE Transaction on Information Theory, **36** (1990), 1334-1380.
8. N. Cesa-Bianchi, Y. Freund, D. Helmbold, and M. K. Warmuth, *On-line prediction and conversion strategies*, Machine Learning, **25** (1996), 71-110.
9. F. Cicalese, U. Vaccaro, *Optimal strategies against a liar*, Theoretical Computer Science, **230** (2000), pp. 167-193.
10. F. Cicalese and D. Mundici, *Optimal binary search with two unreliable tests and minimum adaptiveness*, In: Proc. of ESA99, LNCS vol. **1643**, (1999), 257-266.
11. J. Czyzowicz, D. Mundici, A. Pelc, *Ulam's searching game with lies*, J. Comb. Theor., Ser. A, **52** (1989), 62-76.
12. A. Dhagat, P. Gacs, and P. Winkler, *On Playing "Twenty Question" with a liar*, In: Proc. 3rd ACM-SIAM SODA (1992), 16-22.
13. D.Z. Du, F.K. Hwang, *Combinatorial Group Testing and its Applications*, World Scientific, Singapore, 1993.
14. R. Hill, *Searching with lies*, In: Surveys in Combinatorics, Cambridge Univ. Press (1995), 41-70.
15. R. Karp, *ISIT'98 Plenary Lecture Report: Variations on the theme of 'Twenty Questions'*, IEEE Information Theory Society Newsletter, vol. **49**, No.1, March 1999.
16. C. Kenyon and A. C. Yao, *On Evaluating Boolean Functions with Unreliable Tests*, International J. of Foundation of Computer Science, **1**, (1990), 1-10.
17. F.J. MacWilliams, N.J.A. Sloane, *The Theory of Error-Correcting Codes*, North-Holland, Amsterdam, 1977.
18. D. Mundici, *Ulam's Game, Lukasiewicz Logic and AF  $C^*$ -algebras*, Fundamenta Informaticæ, **18**, 151-161, 1993.
19. D. Mundici, A. Trombetta, *Optimal comparison strategies in Ulam's searching game with two errors*, Theoretical Computer Science, **182**, (1997), 217-232.
20. S. Muthukrishnan, *On optimal strategies for searching in presence of errors*, In: Proc. of the 5th ACM-SIAM SODA (1994), 680-689.
21. A. Negro, M. Sereno, *Ulam's searching game with three lies*, Adv. in Appl. Math., **13** (1992), 404-428.
22. A. Pelc, *Solution of Ulam's problem on searching with a lie*, J. Combin. Theory, Ser. A, **44** (1987), 129-142.
23. A. Pelc, *Searching with permanently faulty tests*, Ars Combinatoria, **38** (1994), 65-76.
24. A. Rényi, *Napló az információelméletről*, Gondolat, Budapest, 1976. (English translation: *A Diary on Information Theory*, J.Wiley and Sons, New York, 1984).
25. R. L. Rivest, A. R. Meyer, D. J. Kleitman, K. Winklmann, J. Spencer, *Coping with errors in binary search procedures*, Proc. of 10th ACM STOC (1978), 227-232.
26. J. Spencer, *Ulam's searching game with a fixed number of lies*, Theoretical Comp. Sci., **95** (1992), 307-321.
27. J. Spencer and P. Winkler, *Three thresholds for a liar*, Combinatorics, Prob. and Comp., **1** (1992), 81-93.
28. A. Tietäväinen, *On the nonexistence of perfect codes over finite fields*, SIAM J. Appl. Math., **24**, (1973), 88-96.
29. S.M. Ulam, *Adventures of a Mathematician*, Scribner's, New York, 1976.
30. J. von Neumann, *Probabilistic Logics and the Synthesis of reliable Organisms from Unreliable Components*, in Automata Studies, Princeton University Press, Princeton, NJ, (1956), 43-98.

# Author Index

- Agarwal, Pankaj K. 328  
Ahal, Shlomo 490  
Alber, Jochen 97  
Aleksandrov, Lyudmil G. 71  
Alstrup, Stephen 46  
Arata, Kouji 300  
Arge, Lars 433, 448  
Arya, Sunil 353  
Asano, Tetsuo 476  
Azar, Yossi 164, 189, 200  
Bender, Michael A. 83  
Berman, Piotr 214  
Berry, Anne 139  
Björklund, Andreas 527  
Bodlaender, Hans L. 97  
Bordat, Jean-Paul 139  
Boyar, Joan 200  
Brodal, Gerth S. 57, 433  
Cheng, Siu-Wing 353  
Cicalese, Ferdinando 549  
Damaschke, Peter 504  
Dinitz, Yefim 272  
Djidjev, Hristo N. 71  
Doddi, Srinivas R. 237  
Epstein, Leah 164, 189  
Favrholdt, Lene M. 200  
Feige, Uriel 10  
Fernau, Henning 97  
Fujishige, Satoru 300  
Gudmundsson, Joachim 314  
Guibas, Leonidas J. 328, 339  
Gupta, Arvind 111  
Har-Peled, Sariel 328  
Hassin, Refael 231, 251  
Heggernes, Pinar 139  
Holm, Jacob 46  
Iacono, John 32  
Irving, Robert W. 259  
Ishii, Toshimasa 286  
Iwata, Satoru 300  
Jacob, Riko 57  
Kalyanasundaram, Bala 150  
Kameda, Tiko 513  
Kao, Ming-Yang 383  
Karger, David R. 1  
Larsen, Kim S. 200  
Levcopoulos, Christos 314  
Liberatore, Vincenzo 175  
Makino, Kazuhisa 300, 513  
Manlove, David F. 259  
Marathe, Madhav V. 237  
Matsui, Tomomi 476  
Mount, David M. 353  
Mukhopadhyaya, Srabani 419  
Mundici, Daniele 549  
Nagamochi, Hiroshi 286  
Narasimhan, Giri 314  
Niedermeier, Rolf 97  
Nielsen, Morten N. 200  
Nishimura, Naomi 111, 125  
Noga, John 150  
Nossenson, Ronit 272  
Pagh, Rasmus 22  
Pagter, Jakob 448  
Palios, Leonidas 367  
Peleg, David 220  
Proskurowski, Andrzej 111  
Pruhs, Kirk 150  
Rabinovich, Yuri 490  
Rabinovitch, Alexander 328  
Ragde, Prabhakar 111, 125  
Ramesh, H. 353  
Ravi, S. S. 237  
Rick, Claus 407  
Rubinstein, Shlomi 231, 251  
Sahni, Sartaj 419  
Samet, Jared 383

- Sandholm, Tuomas 462  
Scott, Sandy 259  
Sethia, Saurabh 83  
Sharir, Micha 328  
Shibuya, Tetsuo 393  
Skiena, Steven 83  
Snoeyink, Jack 339  
Stachowiak, Grzegorz 535  
Stee, Rob van 189  
Sung, Wing-Kin 383  
Suri, Subhash 462  
Taylor, David S. 237  
Thilikos, Dimitrios M. 125  
Thorup, Mikkel 1, 46  
Tokuyama, Takeshi 476  
Toma, Laura 433  
Ukkonen, Esko 20  
Vaccaro, Ugo 549  
Venkataraman, Gayathri 419  
Warkhede, Priyank Ramesh 462  
Widmayer, Peter 237  
Woeginger, Gerhard 150  
Yamashita, Masafumi 513  
Zhang, Li 339